DataNucleus Access Platform v.1.1-SNAPSHOT

Project Documentation

DataNucleus **19 April 2009**

1.1 Welcome

DataNucleus Access Platform 1.1

The DataNucleus Access Platform provides persistence and retrieval of data to a range of datastores using a range of APIs, with a range of query languages.

The majority of competing software only caters for a single API and a single type of datastore. DataNucleus Access Platform is **not just an ORM**. It allows access to RDBMS just like other persistence software, but it also allows access to object-based datastores, or LDAP, or even documents! It is continually evolving. It also provides access via JDO or JPA APIs. It doesn't just stop there. You could define your persistence mapping using JDO (XML or annotations) and then persist using the JPA API. Or define your persistence mapping using JPA (XML or annotations) and then persist using the JDO API!. **Flexibility**

DataNucleus AccessPlatform Checklist

- **License** : Apache 2
- **APIs Supported** : JDO, JPA, REST
- **Datastores Supported** : RDBMS, db4o, LDAP, Excel, XML, NeoDatis, JSON, ODF, AppEngine
- **Query Languages** : JDOQL, JPQL, SQL, db4o Native, NeoDatis Native, NeoDatis Criteria
- **JRE required** : 1.5 or above

If you find something that DataNucleus Access Platform can't handle you can always extend it using its plugin mechanism for one of its defined interfaces.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.2 Getting Started

Access Platform 1.1 : Getting Started

DataNucleus Access Platform implements the JDO and JPA specifications. These specifications define how Java classes can be persisted to a datastore and how they can be queried. By choosing Access Platform you can select which of these APIs you feel most comfortable with. Time for you to get started and use Access Platform!

What is required?

1. **DataNucleus Access Platform 1.1** requires a JDK of 1.5 or above.
2. The first thing to do is Download DataNucleus Access Platform. Download the bundle that is most appropriate to your needs so, for example, if you are going to be developing for access of RDBMS data download the *accessplatform-rdbms* zip. If you want instead to just download the individual DataNucleus plugins that make up Access Platform then please refer to the list of dependencies for details of what other packages are required.
3. You also need a datastore. DataNucleus Access Platform 1.1 supports RDBMS, DB4O, LDAP, Excel spreadsheets, NeoDatis ODB, XML JSON and OpenDocument spreadsheets. Please refer to the linked pages for full details of the datastore required.
4. Depending on what development environment you use, you could also download a DataNucleus plugin for Eclipse or Maven.

That should be enough to get you going. You have the necessary components to start investigating use of DataNucleus.

Starting up

The next thing to do is to learn about JDO and JPA. You need to understand the basic concepts involved. The JDO Overview is one place to start but there is plenty of reading on the internet, starting with the JDO2 or JPA1 specifications of course.

The best thing to do after some reading is to try the JDO Tutorial or the JPA Tutorial. This explains the basic steps of applying JDO/JPA (and DataNucleus) to your own application. The source code from the Tutorial is available for download. Please download it and start up your development environment with the Tutorial classes and files.

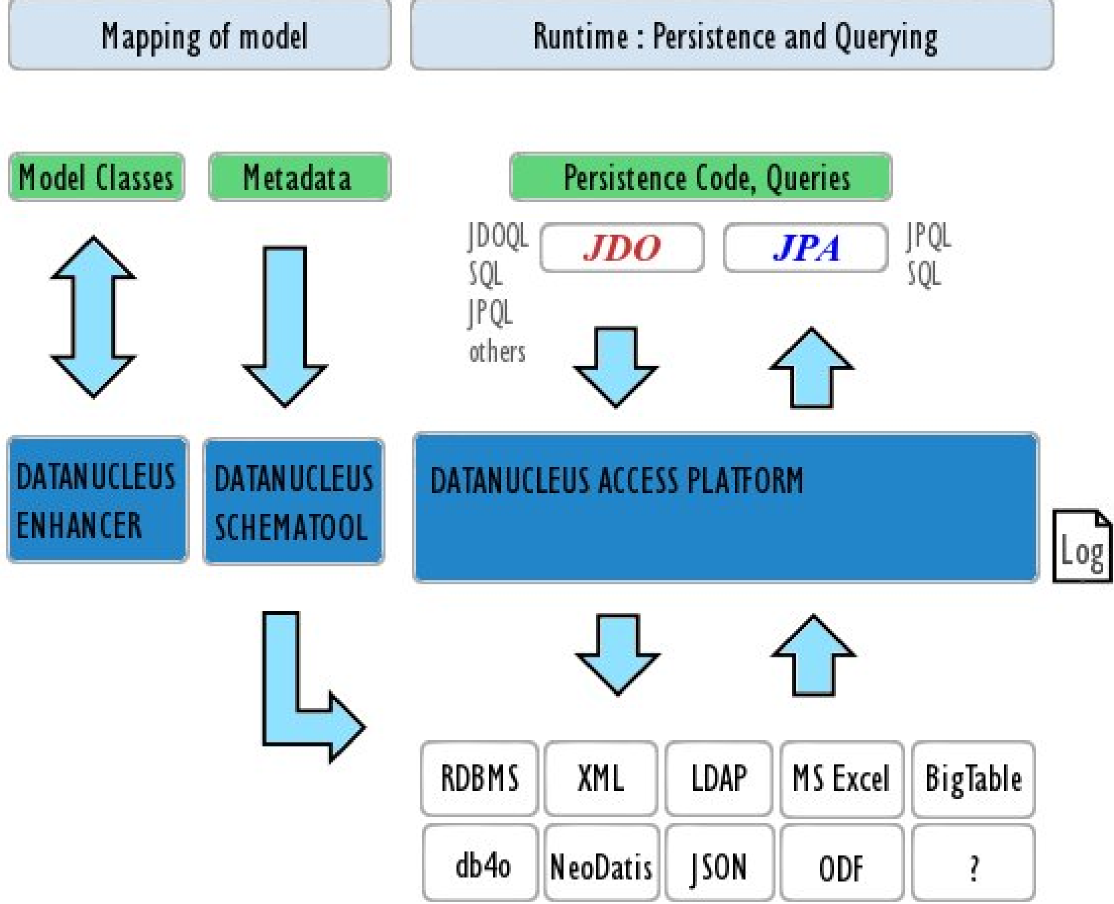
Once you have completed the Tutorial you should be ready to start applying DataNucleus to your own application and benefiting from what it offers.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.3 Development Process

DataNucleus Development Process

DataNucleus attempts to make the whole process of persisting data a transparent process. The idea revolves around the developer having a series of Java classes that need persisting. With DataNucleus, the developer defines the persistence of these classes using Metadata (defined in XML, annotations or by API), and byte-code "enhances" these classes. DataNucleus also provides RDBMS SchemaTool that allows for schema generation/validation before running your application, to make sure that all is correctly mapped. Finally you provide persistence code (to manage the persistence of your objects), and queries (to retrieve your persisted data). DataNucleus Access Platform implements all JDO specifications (1.0, 2.0, 2.1, 2.2 etc) and also the JPA specification (JPA1), plus some preview JPA2 features. *The following diagram shows the process for DataNucleus Access Platform (several parts of the diagram are clickable giving more details).*



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.4 What's New ?

Access Platform : What's New in 1.1

DataNucleus Access Platform version 1.1 extends the 1.0 capabilities aiming for increased stability, performance as well as adding on a more complete feature set for some datastores. Below are some of the new features you can find in DataNucleus Access Platform 1.1.

- JRE 1.5 or above is required to use DataNucleus Access Platform 1.1. If you require JRE 1.3 or 1.4 then please use Access Platform 1.0
- Upgrade db4o support to support db4o 7.0 and above. If you require db4o 6.x please use Access Platform 1.0
- JDO 2.2 : Support for transaction isolation control (previously vendor extension).
- JDO 2.x : Support for "PersistenceManager Proxy"
- JDO 2.3 : Provision of a JDO2.3 compliant enhancer API
- JDO 2.3 : Provision of a JDO2.3 compliant metadata API
- JDO 2.3 : Support for JDO 2.3 Query timeout/cancel API
- JPA : Support for JPQL bulk delete added for DB4O, Excel, JSON, LDAP, NeoDatis and XML datastores
- JPA 2 : Support for @OrderColumn allowing surrogate columns for ordering collections
- Support for Datastore Replication for JDO and JPA
- Add capability to attach objects and update ALL fields not just those that were updated whilst detached (for use in datastore replication).
- RDBMS : Support for sequences with H2 database.
- RDBMS : Start of a rewrite of the SQL generation process
- RDBMS : Support for persistence of interface/Object fields as a single column. Also support for persistence of Collection/Map of interface/Object storing the element as a single column (provides capability to map to Kodo/Xcalia schemas).
- RDBMS : plugin point for defining your own INSERT, UPDATE, DELETE, FETCH and LOCATE operations.
- LDAP : Support for native querying of LDAP datastores (Stefan Seelmann)
- XML : Completed support for 1-1 and 1-N referenced relations
- Support for alternative datastore-identity string form (in particular OpenJPA, Xcalia)
- Addition of MetaData "auto-start mechanism" (Eric Sultan)
- Support input of String forms of ids into pm.getObjectById() for convenience in upgrading from other implementations like Xcalia
- Support for performing enhancement with JDK1.6 *during* compilation
- Support for persisting to OpenDocument spreadsheets

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.5 Upgrade Migration

Access Platform : Migration between versions

DataNucleus Access Platform 1.1 builds on the 1.0 release, adding on more features around the currently supported datastores, and additionally changing the JDK requirement to be JDK1.5 or above. All releases are checked regularly against the JDO/JPA TCKs, meaning that DataNucleus has reached a level of stability in terms of functionality. Occasionally, due to unknown bugs, or due to new functionality being introduced we need to change some aspects of DataNucleus. As a result sometimes users will have to make some changes to move between versions of DataNucleus. We aim to keep this to a minimum.

Migration from 1.1.1 to 1.1.1

Migrating from DataNucleus 1.1.1 to 1.1.2 will require the following changes

- *NucleusJDOHelper.getObjectStateAsString()* has been deprecated for some time and is now removed.

Migration from 1.1.0 to 1.1.1

Migrating from DataNucleus 1.1.0 to 1.1.1 should not require any changes.

Migration from 1.1.0 Milestone4 to 1.1.0 Release

Migrating from DataNucleus 1.1.0 M4 to 1.1.0 Release will require the following changes.

- The value for query timeouts *datanucleus.query.timeout* was previously in seconds, but is now in milliseconds. Adjust values accordingly. This is also now a standard JDO property.

Migration from 1.1.0 Milestone3 to 1.1.0 Milestone4

Migrating from DataNucleus 1.1.0 M3 to 1.1.0 M4 will require the following changes.

- The BCEL enhancer has been removed and you must use ASM now.

Migration from 1.1.0 Milestone2 to 1.1.0 Milestone3

Migrating from DataNucleus 1.1.0 M2 to 1.1.0 M3 will require the following changes.

- The persistence property *datanucleus.datastoreIdentityClassName* is renamed to **datanucleus.datastoreIdentityType**
- The programmatic API for the *DataNucleusEnhancer* has had some changes to align it better with the new JDO2.3 Enhancer API
- The programmatic API for the RDBMS *SchemaTool* has had some changes to allow multiple instances

Migration from 1.1.0 Milestone1 to 1.1.0 Milestone2

Migrating from DataNucleus 1.1.0 M1 to 1.1.0 M2 will require the following changes.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- All "POID" generator classes have been repackaged and renamed. They are now stored in *org.datanucleus.store.valuegenerator*
- Persistence properties relating to "poid" are renamed
    - *datanucleus.poid.transactionIsolation* is now **datanucleus.valuegeneration.transactionIsolation**
    - *datanucleus.poid.transactionAttribute* is now **datanucleus.valuegeneration.transactionAttribute**
- Log category "DataNucleus.POID" is renamed to "DataNucleus.ValueGeneration"

Migration from 1.0.0 Release to 1.1.0 Milestone1

Migrating from DataNucleus 1.0.0 Release to 1.1.0 M1 will require the following changes.

- Upgrade your JDK to be 1.5 or above
- If using db4o, you must now use version 7.0 or above.
- The Java5 plugin has been removed and a JPA plugin added. All JDO annotations are now in "core"
- The "springframework" plugin is no longer required. Specification of transaction isolation can now be performed via the JDO2.2 interface.
- The plugin extension-point *org.datanucleus.java5.annotations* has been renamed to **org.datanucleus.annotations**
- NucleusSQL queries for RDBMS are now no longer supported. Please use straight SQL

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.6 Dependencies

Access Platform : Dependencies

DataNucleus Access Platform utilises some third party software to provide some of its functionality. Dependent on how you intend to use this product you may have to also download some of these third party software packages. You can see below the dependencies and when they are required.

Software	Description	Version	Requirement
JDO API	JDO API definition, developed by Apache JDO.	2.3+	Required
JPA API	JPA API definition	1.0+	Required if you are using the JPA API or JPA annotations
Log4j	Log4J logging library.	1.2+	Required if you wish to log using Log4J. DataNucleus supports Log4J or JDK1.4 logging
ASM	ASM bytecode enhancement framework	3.0+	Required
DB4O	DB4O object database	7.0+	Required if you are using a DB4O datastore
NeoDatis	NeoDatis object database	1.9.0-beta-3.9+	Required if you are using a NeoDatis datastore
Apache POI	Apache library for writing to Microsoft documents	3.2	Required if you want to use Excel documents
Oracle Coherence	Oracle Coherence caching product		Required if you want to use Oracle Coherence for level 2 caching. This is commercial software
EHCache	EHCache caching product	1.0, 1.1	Required if you want to use EHCache for level 2 caching
OSCache	OSCache caching product	2.1	Required if you want to use OSCache for level 2 caching
SwarmCache	SwarmCache caching product	1.0RC2	Required if you want to use SwarmCache for level 2 caching
C3P0	C3P0 RDBMS connection pooling library	0.9.0+	Required if you are using an RDBMS datastore and want to use C3P0 for connection pooling
commons-dbcp	DBCP RDBMS connection pooling library	1.1+	Required if you are using an RDBMS datastore and want to use DBCP for connection pooling
commons-pool	DBCP RDBMS connection pooling library	1.1+	Required if you are using an RDBMS datastore and want to use DBCP for connection pooling
commons-collections	Apache commons collections library	3.0+	Required if you are using an RDBMS datastore and want to use DBCP for connection pooling

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Software	Description	Version	Requirement
proxool	Proxool RDBMS connection pooling library	0.9.0RC3	Required if you are using an RDBMS datastore and want to use Proxool for connection pooling
commons-logging	Apache commons logging library	1.0+	Required if you are using an RDBMS datastore and want to use Proxool for connection pooling
Flexjson	Flexjson	1.7+	Required if you want to use the REST API.
mx4j	MX4J management library	3.0+	Required if you want to manage DataNucleus operations using MX4J
jaxb-api	2.1	JAXB API	Required is you are using an XML datastore
jaxb-impl	2.x	JAXB Implementation	Required is you are using an XML datastore
mx4j-tools	MX4J tools	1.2+	Required if you want to manage DataNucleus operations using MX4J
sdoapi	Oracle Spatial library	1.2+	Required if you want to persist Oracle spatial types
JDBC Driver	JDBC Driver for your chosen RDBMS		Required if you want to use an RDBMS datastore. Obtain from your RDBMS vendor
ODFDOM	ODF Toolkit for Java	0.6+	Required if you want to use an ODF document for persistence.
Xerces	Xerces XML parser	2.8+	Required if you want to use an ODF document for persistence. Required by ODFDOM

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.7 Architecture

Access Platform : Architecture

The DataNucleus Access Platform provides persistence and retrieval of data to a range of datastores using a range of APIs, with a range of query languages. It is architected with flexibility in mind.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.8 Compatibility

Access Platform : Compatibility

We aim to make DataNucleus AccessPlatform as compatible with related software as possible. Here we give an overview of known compatibilities/problems

Software	Status
GraniteDS	The GraniteDS team wrote a plugin to support DataNucleus. This is in their 2.0.0.B1 release and later.
Scala	In versions up to and including 1.1.1 there was a problem since Scala generates constructor bytecode that doesn't always initialise the superclass first. This was fixed in 1.1.2 (datanucleus-enhancer)
GWT	There is a known problem in the serialisation of detached objects where GWT fails to handle serialisation of a field. This is a problem in GWT since the field is of type Object[] and every element is Serializable. There is a project GILEAD that attempts to handle this for various persistence solutions. Also look at this and this.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1.9 JDO-JPA FAQ

JDO - JPA Frequently Asked Questions

DataNucleus Access Platform supports both JDO and JPA specifications of Java persistence. As such it has no "vested interest" in either technology, believing that it is for users to choose which they like best. There has been much FUD on the web about JDO and JPA, largely perpetrated by RDBMS vendors. This FAQ corrects many of these points

Q: Which specification was the original?
JDO was the first Java persistence specification, starting in 1999, and the JDO 1.0 specification being published in April 2002. This provided the persistence API, and was standardised as JSR012. In May 2006 JDO 2.0 was released. This provided an update to the persistence API as well as a complete definition of ORM, standardised as JSR243. Later in May 2006 JPA 1.0 was released. This provided a persistence API, and a limited definition of ORM, concentrating only on RDBMS, and was standardised as JSR220.

Q: Why did JPA come about when we already had a specification for Java persistence in JDO?
Politics. RDBMS vendors didn't like the idea of having a technology that allowed users to leverage a single API, and easily swap to a different type of datastore. Much pressure was applied to SUN to provide a different specification, and even to try to say that JPA was to supercede JDO. The JCP is dominated by large organisations and SUN capitulated. They even published a FAQ to try to justify their decision.

Q: Is JDO dead?
No. As part of SUN's capitulation above, they donated JDO to Apache to develop the technology further. In the last year and a half there have been two revisions to the JDO 2 specification;

- JDO 2.1 adding on support for annotations, enums, and some JPA concepts.
- JDO 2.2 adding on support for dynamic fetch groups, transaction isolation and cache control.

Q: Will JPA replace JDO ?
It is hard to see that happening since JPA provides nothing to cater for persistence of Java objects to non-RDBMS datastores. It doesn't even provide a complete definition of ORM, so cannot yet compete with JDO's ORM handling. Perhaps in JPA 2.0 (2009?) it will be able to cope with the majority of things that JDO does for RDBMS. JDO is still being developed, and while users require this technology then it will continbue to exist. *DataNucleus will continue to support both APIs since there is a need for both in modern enterprise applications*

Q: What differences are there between how JDO is developed and how JPA is developed ?
JPA is developed in private by an "expert group". JDO is developed in public by anybody interested in the technology. The tests to verify compliance with JPA are only available after signing non-disclosure agreements with SUN and this process can take up to 3 months just to get the test suite. The tests to verify compliance with JDO are freely downloadable and can be run by users or developers. This means that anybody can check whether an implementation is compliant with JDO, whereas the same is not true of JPA. *DataNucleus run the JDO 2 and JPA 1 TCKs at frequent intervals and publish the results on our website.*

Q: Why should I use JDO when JPA is supported by "large organisations" ?
By "large organisations" you presumably mean commercial organisations like Oracle, SUN, RedHat(JBoss). And they have their own vested interest in RDBMS technologies, or in selling application

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

servers. You should make your own decisions rather than just follow down the path you are shepherded in by any commercial organisation. Your application will be **supported by you** not by them. The technology you use should be the best for the job and what you feel most comfortable with. If you feel more comfortable with JPA and it provides all that your application needs then use it. Similarly if JDO provides what you need then you use that. For this reason DataNucleus provides support for both specifications.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.1 Bytecode Enhancement

Class enhancement

DataNucleus requires that all classes that are persisted implement *PersistenceCapable*, an interface defined by JDO. **Why should we do this, Hibernate/TopLink dont need it ?**. Well thats a simple question really

- DataNucleus uses this *PersistenceCapable* interface, and adds it using bytecode enhancement techniques so that you never need to actually change your classes. This means that you get transparent persistence, and your classes always remain *your* classes. ORM tools that use a mix of reflection and/or proxies are not totally transparent.
- DataNucleus' use of *PersistenceCapable* provides transparent change tracking. When any change is made to an object the change creates a notification to DataNucleus allowing it to be optimally persisted. ORM tools that dont have access to such change tracking have to use reflection to detect changes. The performance of this process will break down as soon as you read a large number of objects, but modify just a handful, with these tools having to compare all object states for modification at transaction commit time.

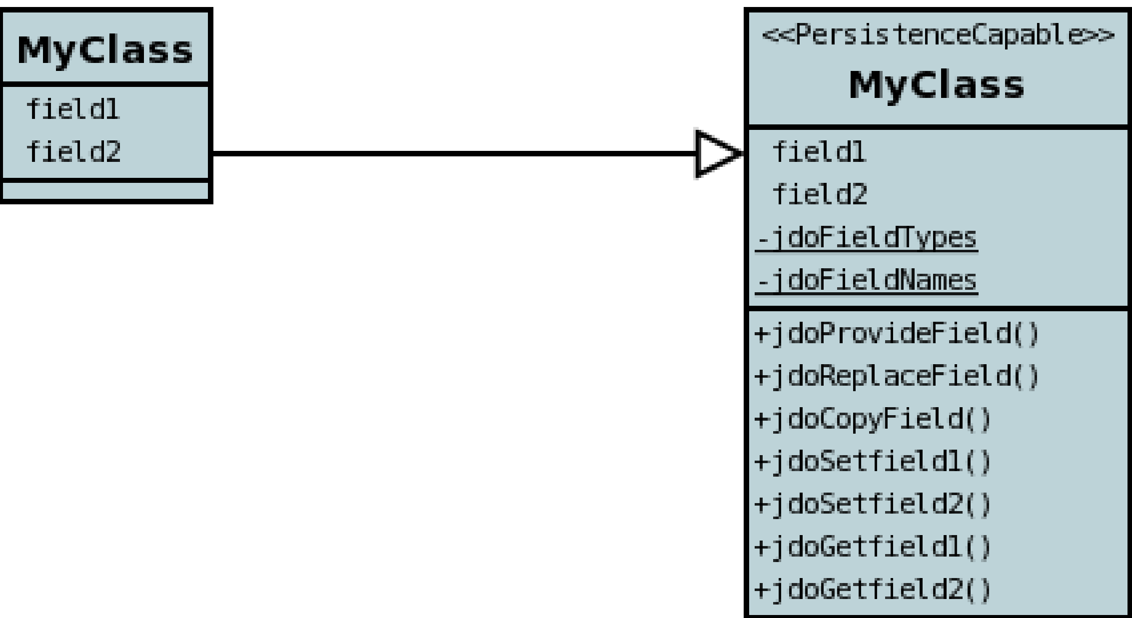
In a JDO-enabled application there are 3 categories of classes. These are PersistenceCapable, PersistenceAware and normal classes. The Meta-Data defines which classes fit into these categories. To give an example for JDO, we have 3 classes. The class A is to be persisted in the datastore. The class B directly updates the fields of class A but doesn't need persisting. The class C is not involved in the persistence process. We would define JDO MetaData for these classes like this

```
<class name="A" persistence-modifier="persistence-capable">
    <field name="myField">
        ...
    </field>
    ...
</class>
<class name="B" persistence-modifier="persistence-aware">
</class>
```

So our MetaData is mainly for those classes that are PersistenceCapable and are to be persisted to the datastore (we don't really need the persistence-modifier for thse classes since this is the default). For PersistenceAware classes we simply notate that the class knows about persistence. We don't define MetaData for any class that has no knowledge of persistence.

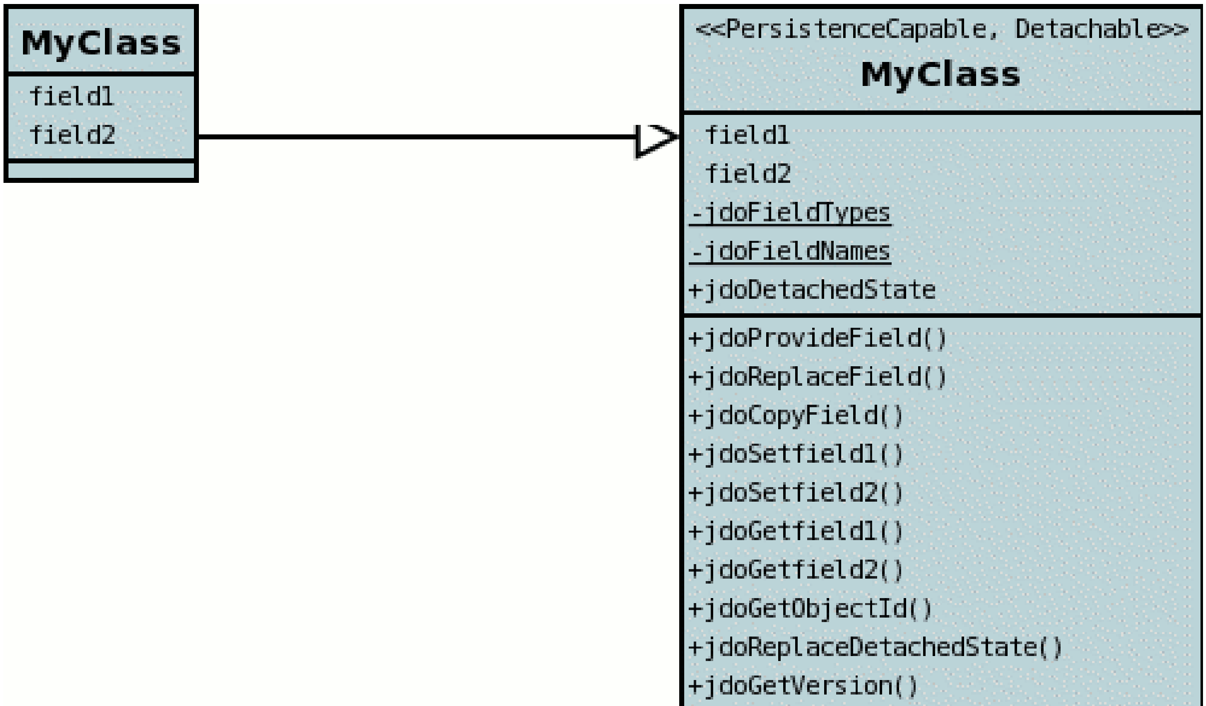
JDO requires that all classes to be persisted must implement the PersistenceCapable interface . Users could manually do this themselves but this would impose work on them. JDO permits the use of a byte-code enhancer that converts the users normal classes to implement this interface. DataNucleus provides its own byte-code enhancer (this can be found in the datanucleus-enhancer.jar). This section describes how to use this enhancer with DataNucleus. The DataNucleus enhancer fully implements JDO2 and so is the recommended choice when persisting using the JDO2 API. The enhancement process adds the necessary methods to the users class in order to implement PersistenceCapable.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The example above doesn't show all PersistenceCapable methods, but demonstrates that all added methods and fields are prefixed with "jdo" to distinguish them from the users own methods and fields. Also each persistent field of the class will be given a jdoGetXXX, jdoSetXXX method so that accesses of these fields are intercepted so that JDO can manage their "dirty" state.

The MetaData defines which classes are required to be persisted, and also defines which aspects of persistence each class requires. For example if a class has the detachable attribute set to true, then that class will be enhanced to also implement Detachable



Again, the example above doesn't show all methods added for the Detachable interface but the main thing to know is that the detached state (object id of the datastore object, the version of the datastore

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

object when it was detached, and which fields were detached is stored in "jdoDetachedState"). Please see the JDO spec for more details.

If the MetaData is changed in any way during development, the classes should always be recompiled and re-enhanced afterwards.

Byte-Code Enhancement Myths

Some groups (e.g Hibernate) perpetuated arguments against "byte-code enhancement" saying that it was somehow 'evil'. The most common were :-

- Slows down the code-test cycle. This is erroneous since you only need to enhance just before test and the provided plugins for Ant, Eclipse and Maven all do the enhancement job automatically and rapidly.
- Is less "lazy" than the proxy approach since you have to load the object as soon as you get a pointer to it. In a 1-1 relation you have to load the object then since you would cause issues with null pointers otherwise. With 1-N relations you load the elements of the collection/map only when you access them and not the collection/map. Hardly an issue then is it!
- Fail to detect changes to public fields unless you enhance your client code. Firstly very few people will be writing code with public fields since it is bad practice in an OO design, and secondly, this is why we have "PersistenceAware" classes.

So as you can see, there are no valid reasons against byte-code enhancement, and the pluses are that runtime detection of dirty events on objects is much quicker, hence your persistence layer operates faster without any need for iterative reflection-based checks. The fact is that Hibernate itself also now has a mode whereby you can do bytecode enhancement although not the default mode of Hibernate. So maybe it wasn't so evil after all ?

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.2 Enhancer

DataNucleus Enhancer

As is described in the ByteCode Enhancement guide, DataNucleus utilises the common technique of byte-code manipulation to make your normal Java classes "persistable". The mechanism provided by DataNucleus is to use an "enhancer" process to perform this manipulation before you use your classes at runtime. The process is very quick and easy.

How to use the DataNucleus Enhancer depends on what environment you are using. Below are some typical examples.

- Automatic invocation from javac
- Manual invocation at the command line
- Using Maven1 via the DataNucleus Maven1 plugin
- Using Maven2 via the DataNucleus Maven2 plugin
- Using Ant
- Runtime Enhancement
- Using the Eclipse DataNucleus plugin
- Programmatically via an API

Javac

DataNucleus provides a JAR containing the Enhancer (datanucleus-enhancer.jar). From J2SE 1.6 and forward, you can automatically enhance your classes when compiling your classes. Just add the datanucleus-enhancer.jar, datanucleus-cor.jar, jdo2-api.jar and asm.jar to the compiler classpath and the classes will be enhanced.

Manually

DataNucleus provides a JAR containing the Enhancer (datanucleus-enhancer.jar). If you are building your application manually and want to enhance your classes you follow the instructions in this section. You invoke the enhancer as follows

```
java -cp classpath  org.datanucleus.enhancer.DataNucleusEnhancer [options]
[jdo-files] [class-files]
    where options can be
        -persistenceUnit persistence-unit-name : Name of a "persistence-unit" to
enhance the classes for
        -d target-dir-name : Write the enhanced classes to the specified directory
        -api api-name : Name of the API we are enhancing for (JDO, JPA). Default is
JDO
        -enhancerName name : Name of the ClassEnhancer to use. Options ASM
        -checkonly : Just check the classes for enhancement status
        -v : verbose output

    where classpath must contain the following
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        datanucleus-enhancer.jar
        datanucleus-core.jar
        asm.jar
        jdo2-api.jar
        log4j.jar (optional)
        your classes
        your meta-data files
```

The input to the enhancer should be *either* a set of MetaData/class files *or* the name of the "persistence-unit" to enhance. In the first option, if any classes have annotations then they must be specified. All classes and MetaData files should be in the CLASSPATH when enhancing. To give an example of how you would invoke the enhancer

```
Linux/Unix :
java -cp
target/classes:lib/datanucleus-enhancer.jar:lib/datanucleus-core.jar:lib/jdo2-api.jar:
                    lib/log4j.jar:lib/asm.jar
     -Dlog4j.configuration=file:log4j.properties
     org.datanucleus.enhancer.DataNucleusEnhancer
     **/*.jdo

Windows :
java -cp
target\classes;lib\datanucleus-enhancer.jar;lib\datanucleus-core.jar;lib\jdo2-api.jar;
                    lib\log4j.jar;lib\asm.jar
     -Dlog4j.configuration=file:log4j.properties
     org.datanucleus.enhancer.DataNucleusEnhancer -v
     target/classes/org/mydomain/mypackage1/package.jdo
     target/classes/org/mydomain/mypackage2/package.jdo

[should all be on same line. Shown like this for clarity]
```

So you pass in your JDO MetaData files (and/or the class files wihich use annotations) as the final argument(s) in the list, and include the respective JAR's in the classpath (-cp). The enhancer responds as follows

```
DataNucleus Enhancer (version 1.0.0) : Enhancement of classes

DataNucleus Enhancer : Classpath
>>  /home/andy/work/myproject//target/classes
>>  /home/andy/work/myproject/lib/log4j.jar
>>  /home/andy/work/myproject/lib/jdo2-api.jar
>>  /home/andy/work/myproject/lib/datanucleus-core.jar
>>  /home/andy/work/myproject/lib/datanucleus-enhancer.jar
>>  /home/andy/work/myproject/lib/asm.jar

DataNucleus Enhancer : Using ClassEnhancer "asm" for API "JDO"

DataNucleus Enhancer : Input Files
>>  /home/andy/work/myproject/target/classes/org/mydomain/mypackage1/package.jdo
>>  /home/andy/work/myproject/target/classes/org/mydomain/mypackage2/package.jdo

Processing class "org.mydomain.mypackage1.Pack"
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
ENHANCED: org.mydomain.mypackage1.Pack
Processing class "org.mydomain.mypackage1.Card"
ENHANCED: org.mydomain.mypackage1.Card
Processing class "org.mydomain.mypackage2.Pack"
ENHANCED: org.mydomain.mypackage2.Pack
Processing class "org.mydomain.mypackage2.Card"
ENHANCED: org.mydomain.mypackage2.Card
DataNucleus Enhancer completed with success for 4 classes. Timings : input=422 ms,
enhance=490 ms, total=912 ms.
     ... Consult the log for full details
```

If you have errors here relating to "Log4J" then you must fix these first. If you receive no output about which class was ENHANCED then you should look in the DataNucleus enhancer log for errors. The enhancer performs much error checking on the validity of the passed MetaData and the majority of errors are caught at this point. You can also use the DataNucleus Enhancer to check whether classes are enhanced. To invoke the enhancer in this mode you specify the checkonly flag. This will return a list of the classes, stating whether each class is enhanced for persistence under JDO or not. The classes need to be in the CLASSPATH *(Please note that a CLASSPATH should contain a set of JAR's, and a set of directories. It should NOT explictly include class files, and should NOT include parts of the package names. If in doubt please consult a Java book).*

Maven1

Maven1 operates from a series of plugins. There is a DataNucleus plugin for Maven1 that allows enhancement of classes. Go to the Download section of the website and download this. Once you have the Maven1 plugin, you then need to set the properties for the plugin in your project.properties file. This will typically not require any addition to your project.properties. If you do need to change this file, the following parameters are the likely ones to change

```
maven.datanucleus.jdo.fileset.dir=${maven.build.dest}  # Location of the JDO files
maven.datanucleus.jdo.fileset.include=**/*.jdo         # fileset to include
#maven.datanucleus.jdo.fileset.exclude=something.jdo   # fileset to exclude, if any

maven.datanucleus.classes.dir=${maven.build.dest}      # Location of classes to
enhance
maven.datanucleus.api=JDO                              # API to enhance to (JDO,
JPA)
maven.datanucleus.enhancer.classenhancer=asm           # Use ASM

maven.datanucleus.inputmode=files                      # Mode of input. Can also be
set to "persistenceunit"
maven.datanucleus.persistenceunit=                     # Name of the
persistence-unit to enhance
maven.datanucleus.log4j.configuration=                 # Log definition to use
maven.datanucleus.verbose=true                         # Turn on more output ?
```

You then run the Maven DataNucleus plugin, as follows

```
maven datanucleus:enhance
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This will enhance all classes found that correspond to the classes defined in the JDO files in your source tree. If you want to check the current status of enhancement you can also type

```
maven datanucleus:enhance-check
```

Maven2

Maven2 operates from a series of plugins. There is a DataNucleus plugin for Maven2 that allows enhancement of classes. Go to the Download section of the website and download this. Once you have the M2 plugin, you then need to set any properties for the plugin in your *pom.xml* file. If you are using annotations then you'll need to add **.class* to "mappingIncludes" for example. Some properties that you may need to change are below

Property	Default	Description
mappingIncludes	**/*.jdo	Fileset to include for enhancement
mappingExcludes		Fileset to exclude for enhancement
persistenceUnitName		Name of the persistence-unit to enhance
log4jConfiguration		Config file location for Log4J (if using it)
jdkLogConfiguration		Config file location for JDK1.4 logging (if using it)
api	JDO	API to enhance to (JDO, JPA)
verbose	false	Verbose output?
targetDirectory		Where the enhanced classes are written (default is to overwrite them)
fork	true	Whether to fork the enhancer process

You then run the Maven2 DataNucleus plugin, as follows

```
mvn datanucleus:enhance
```

This will enhance all classes found that correspond to the classes defined in the JDO files in your source tree. If you want to check the current status of enhancement you can also type

```
mvn datanucleus:enhance-check
```

Or alternatively, you could add the following to your POM

```
    <build>
        ...
        <plugins>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <plugin>
                <groupId>org.datanucleus</groupId>
                <artifactId>datanucleus-maven-plugin</artifactId>
                <version>1.1.0</version>
                <configuration>
<log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
                    <verbose>true</verbose>
                </configuration>
                <executions>
                    <execution>
                        <phase>compile</phase>
                        <goals>
                            <goal>enhance</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
        ...
    </build>
```

So you then get auto-enhancement after each compile

Ant

Ant provides a powerful framework for performing tasks. DataNucleus provides an Ant task to enhance classes. DataNucleus provides a JAR containing the Enhancer (datanucleus-enhancer.jar). You need to make sure that the datanucleus-enhancer.jar, datanucleus-core.jar, asm.jar, log4j.jar and jdo2-api.jar are in your classpath. In the DataNucleus Enhancer Ant task, the following parameters are available

Parameter	Description	values
dir	Optional. Directory containing the JDO files to use for enhancing. Uses ant build file directory if the parameter is not specified.	
destination	Optional. Defining a directory where enhanced classes will be written. If omitted, the original classes are updated.	
api	Optional. Defines the API to be used when enhancing	**JDO**, JPA
enhancerName	Optional. Defines the ClassEnhancer to use when enhancing.	**ASM**
persistenceUnit	Optional. Defines the "persistence-unit" to enhance.	
checkonly	Whether to just check the classes for enhancement status. Will respond for each class with "ENHANCED" or "NOT ENHANCED". **This will disable the enhancement process and just perform these checks.**	true, false
verbose	Whether to have verbose output.	true, false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Parameter	Description	values
filesuffixes	Optional. Suffixes to accept for the input files. The Enhancer Ant Task will scan for the files having these suffixes under the directory specified by *dir* option. The value can include comma-separated list of suffixes. If using annotations you can have "class" included as a valid suffix here or use the *fileset*.	jdo
fileset	Optional. Defines the files to accept as the input files. Fileset enables finer control to which classes / metadata files are accepted to enhanced. If one or more files are found in the fileset, the Enhancer Ant Task will not scan for additional files defined by the option *filesuffixes*. For more information on defining a fileset, see Apache FileSet Manual.	
if	Optional. The name of a property that must be set in order to the Enhancer Ant Task to execute.	

The enhancer task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the enhancer task.

So you could define something *like* the following, setting up the parameters **enhancer.classpath**, **jdo.file.dir**, and **log4j.config.file** to suit your situation (the **jdo.file.dir** is a directory containing the JDO files defining the classes to be enhanced). The classes specified by the XML Meta-Data files, together with the XML Meta-Data files must be in the CLASSPATH *(Please note that a CLASSPATH should contain a set of JAR's, and a set of directories. It should NOT explictly include class files, and should NOT include parts of the package names. If in doubt please consult a Java book).*

```
<target name="enhance" description="DataNucleus enhancement">
    <taskdef name="datanucleusenhancer" classpathref="enhancer.classpath"
                classname="org.datanucleus.enhancer.tools.EnhancerTask" />

    <datanucleusenhancer classpathref="enhancer.classpath"
        dir="${jdo.file.dir}" failonerror="true" verbose="true">
        <jvmarg line="-Dlog4j.configuration=${log4j.config.file}"/>
    </datanucleusenhancer>
</target>
```

You can also define the files to be enhanced using a **fileset**. When a **fileset** is defined, the Enhancer Ant Task will not scan for additional files, and the option *filesuffixes* is ignored.

```
<target name="enhance" description="DataNucleus enhancement">
    <taskdef name="datanucleusenhancer" classpathref="enhancer.classpath"
                classname="org.datanucleus.enhancer.tools.EnhancerTask" />

    <datanucleusenhancer
        dir="${jdo.file.dir}" failonerror="true" verbose="true">
        <fileset dir="${classes.dir}">
            <include name="**/*.jdo"/>
            <include name="**/acme/annotated/persistentclasses/*.class"/>
        </fileset>
        <classpath>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <path refid="enhancer.classpath"/>
        </classpath>
    </datanucleusenhancer>
</target>
```

You can disable the enhancement execution upon the existence of a property with the usage of the *if* parameter.

```
<target name="enhance" description="DataNucleus enhancement">
    <taskdef name="datanucleusenhancer" classpathref="enhancer.classpath"
                classname="org.datanucleus.enhancer.tools.EnhancerTask"
if="aPropertyName"/>

    <datanucleusenhancer classpathref="enhancer.classpath"
        dir="${jdo.file.dir}" failonerror="true" verbose="true">
        <jvmarg line="-Dlog4j.configuration=${log4j.config.file}"/>
    </datanucleusenhancer>
</target>
```

Runtime Enhancement

Enhancement of persistent classes at runtime is possible when using JRE 1.5 or superior versions. Runtime Enhancement requires the following runtime dependencies: ASM, DataNucleus Core and DataNucleus Enhancer libraries. To enable runtime enhancement, the *javaagent* option must be set in the java command line. Example:

```
java -javaagent:datanucleus-enhancer-1.0-SNAPSHOT.jar Main
```

The statement above will mean that all classes, when being loaded, will be processed by the ClassFileTransformer (except class in packages "java.*", "javax.*", "org.datanucleus.*"). This means that it can be slow since the MetaData search algorithm will be utilised for each. To speed this up you can specify an argument to that command specifying the names of package(s) that should be processed (and all others will be ignored). Like this

```
java
-javaagent:datanucleus-enhancer-1.0-SNAPSHOT.jar=mydomain.mypackage1,mydomain.mypackage2
Main
```

so in this case only classes being loaded that are in *mydomain.mypackage1* and *mydomain.mypackage2* will be attempted to be enhanced.

Please take care over the following when using runtime enhancement

- When you have a class with a field of another persistable type make sure that you mark that field as

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

"persistent" (@Persistent, or in XML) since with runtime enhancement at that point the related class is likely not yet enhanced so will likely not be marked as persistent otherwise. **Be explicit**

Programmatic API

You could alternatively programmatively enhance classes from within your application. This is done as follows

```
import javax.jdo.JDOEnhancer;

JDOEnhancer enhancer = JDOHelper.getEnhancer();
enhancer.setVerbose(true);
enhancer.addPersistenceUnit("MyPersistenceUnit");
enhancer.enhance();
```

This will look in META-INF/persistence.xml and enhance all classes defined by that unit.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.3 Persistence Properties

Persistence Properties

Any JDO-enabled application will require at least one PersistenceManagerFactory which accepts properties to define its capabilities. Any JPA-enabled application will require at least one EntityManagerFactory which also accepts properties to define its capabilities. DataNucleus provides a large number of properties for use with either JDO or JPA APIs.

Use of the following properties gives you more control over the operations of DataNucleus, but bear in mind that these properties are only for use with DataNucleus and will not work with other JDO/JPA implementations.

- Datastore Definition - datastore properties
- General - general properties
- Schema Control - properties controlling the generation of the datastore schema.
- Transactions and Locking - properties controlling how transactions operate
- Caching - properties controlling the behaviour of the cache(s)
- Value Generation - properties controlling the generation of object identities and field values
- MetaData - metadata properties
- Auto-Start - Auto-Start Mechanism properties
- Query - properties controlling the behaviour of queries
- JPA - properties allowing extra functionality with JPA.

Please note that there are additional persistence properties for each supported datastore. See RDBMS and DB4O

Datastore Definition

datanucleus.ConnectionFactory	
Description	Instance of a connection factory. For RDBMS, it must be an instance of javax.sql.DataSource. See Data Sources. **This is for a transactional DataSource**
Range of Values	

datanucleus.ConnectionFactory2	
Description	Instance of a connection factory. For RDBMS, it must be an instance of javax.sql.DataSource. See Data Sources. **This is for a non-transactional DataSource**
Range of Values	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.ConnectionFactoryName

Description	The JNDI name for a connection factory. For RBDMS, it must be a JNDI name that points to a javax.sql.DataSource object. See Data Sources. **This is for a transactional DataSource**
Range of Values	

datanucleus.ConnectionFactory2Name

Description	The JNDI name for a connection factory. For RBDMS, it must be a JNDI name that points to a javax.sql.DataSource object. See Data Sources. **This is for a non-transactional DataSource**
Range of Values	

datanucleus.ConnectionDriverName

Description	The name of the (JDBC) driver to use for the DB
Range of Values	

datanucleus.ConnectionURL

Description	URL specifying the datastore to use for persistence
Range of Values	

datanucleus.ConnectionUserName

Description	Username to use for connecting to the DB
Range of Values	

datanucleus.ConnectionPassword

Description	Password to use for connecting to the DB
Range of Values	

General

datanucleus.IgnoreCache

Description	Whether to ignore the cache for queries
Range of Values	true | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.Multithreaded

Description	Whether to run the PersistenceManager multithreaded. **Nit that this is a hint only to try to allow thread-safe operations on the PM**
Range of Values	true | false

datanucleus.NontransactionalRead

Description	Whether to allow nontransactional reads
Range of Values	true | false

datanucleus.NontransactionalWrite

Description	Whether to allow nontransactional writes
Range of Values	true | false

datanucleus.Optimistic

Description	Whether to use Optimistic transactions For JDO this defaults to *false* and for JPA it defaults to *true*
Range of Values	true | false

datanucleus.RetainValues

Description	Whether to suppress the clearing of values from persistent instances on transaction completion
Range of Values	true | false

datanucleus.RestoreValues

Description	Whether persistent object have transactional field values restored when transaction rollback occurs.
Range of Values	true | false

datanucleus.Mapping

Description	Name for the ORM MetaData mapping files to use with this PMF. For example if this is set to "mysql" then the implementation looks for MetaData mapping files called "{classname}-mysql.orm" or "package-mysql.orm". If this is not specified then the JDO implementation assumes that all is specified in the JDO MetaData file. *RDBMS datastores only*
Range of Values	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.mapping.Catalog

Description	Name of the catalog to use by default for all classes persisted using this PMF/EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this catalog name if the RDBMS supports specification of catalog names in DDL. *RDBMS datastores only*
Range of Values	

datanucleus.mapping.Schema

Description	Name of the schema to use by default for all classes persisted using this PMF/EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this schema name if the RDBMS supports specification of schema names in DDL. *RDBMS datastores only*
Range of Values	

datanucleus.DetachAllOnCommit

Description	Allows the user to select that when a transaction is committed all objects enlisted in that transaction will be automatically detached.
Range of Values	true | false

datanucleus.CopyOnAttach

Description	Whether, when attaching a detached object, we create an attached copy or simply migrate the detached object to attached state
Range of Values	true | false

datanucleus.TransactionType

Description	Type of transaction to use. If running under J2SE the default is RESOURCE_LOCAL, and if running under J2EE the default is JTA.
Range of Values	RESOURCE_LOCAL | JTA

datanucleus.ServerTimeZoneID

Description	Id of the TimeZone under which the datastore server is running. If this is not specified or is set to null it is assumed that the datastore server is running in the same timezone as the JVM under which DataNucleus is running.
Range of Values	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.PersistenceUnitName

Description	Name of a *persistence-unit* to be found in a *persistence.xml* file (under META-INF) that defines the persistence properties to use and the classes to use within the persistence process.
Range of Values	

datanucleus.persistenceXmlFilename

Description	URL name of the *persistence.xml* file that should be used instead of using "META-INF/persistence.xml".
Range of Values	

datanucleus.storeManagerType

Description	Type of the StoreManager to use for this PMF/EMF. This has typical values of "rdbms", "db4o". If it isnt specified then it falls back to trying to find the StoreManager from the connection URL. The associated DataNucleus plugin has to be in the CLASSPATH when selecting this. When using data sources (as usually done in a JavaEE container), DataNucleus cannot find out the correct type automatically and this option must be set.
Range of Values	rdbms | db4o | alternate StoreManager key

datanucleus.managedRuntime

Description	Whether to allow management of the runtime of DataNucleus. Allows hooking in JMX. Please refer to the Management Guide
Range of Values	true | **false**

datanucleus.deletionPolicy

Description	Allows the user to decide the policy when deleting objects. The default is "JDO2" which firstly checks if the field is dependent and if so deletes dependents, and then for others will null any foreign keys out. The problem with this option is that it takes no account of whether the user has also defined <foreign-key> elements, so we provide a "DataNucleus" mode that does the dependent field part first and then if a FK element is defined will leave it to the FK in the datastore to perform any actions, and otherwise does the nulling.
Range of Values	JDO2 | DataNucleus

datanucleus.findObjectCheckInheritance

Description	When retrieving an object by identity DataNucleus can do a check on the inheritance level of the object. In many cases this check is not adding anything since the id implies the inheritance level. This allows the user to turn off the check.
Range of Values	true | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.identityTranslatorType

Description	You can allow identities input to *pm.getObjectById* be translated into valid JDO ids if there is a suitable translator. See Identity Translator Plugin
Range of Values	

datanucleus.datastoreIdentityType

Description	Which "datastore-identity" class plugin to use to represent datastore identities. Refer to Datastore Identity extensions for details.
Range of Values	datanucleus | kodo | xcalia | {user-supplied plugin}

datanucleus.attachSameDatastore

Description	When attaching an object DataNucleus by default makes no assumption about which datastore the object was detached from and so makes a check for existence before attaching each object. This option allows you to turn off that check when you know you are detaching and attaching using the same datastore.
Range of Values	false | true

datanucleus.attachPolicy

Description	When attaching the default behaviour is to attach fields/properties that have been changed since being detached. This property allows changing of this behaviour so that if using "attach-all" all fields are attached.
Range of Values	attach-dirty | attach-all

datanucleus.detachAsWrapped

Description	When detaching, any mutable second class objects (Collections, Maps, Dates etc) are typically detached as the basic form (so you can use them on client-side of your application). This property allows you to select to detach as wrapped objects. It only works with "detachAllOnCommit" situations (not with detachCopy) currently
Range of Values	true | false

datanucleus.detachOnClose

Description	This allows the user to specify whether, when a PersistenceManager is closed, that all objects in the L1 cache are automatically detached. **Users are recommended to not use this option, and instead use the JDO2 standard *datanucleus.DetachAllOnCommit***. This option may be removed in a later release.
Range of Values	false | true

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.detachmentFields

Description	When detaching you can control what happens to loaded/unloaded fields of the FetchPlan. The default for JDO is to load any unloaded fields of the current FetchPlan before detaching. You can also unload any loaded fields that are not in the current FetchPlan (so you only get the fields you require) as well as a combination of both options
Range of Values	load-fields | unload-fields | load-unload-fields

datanucleus.manageRelationships

Description	This allows the user control over whether DataNucleus will try to manage bidirectional relations, correcting the input objects so that all relations are consistent. This process runs when flush()/commit() is called. You can set it to *false* if you always set both sides of a relation when persisting/updating.
Range of Values	true | false

datanucleus.manageRelationshipsChecks

Description	This allows the user control over whether DataNucleus will make consistency checks on bidirectional relations. If "datanucleus.managedRelationships" is not selected then no checks are performed. If a consistency check fails at flush()/commit() then a JDOUserException is thrown. You can set it to *false* if you want to omit all consistency checks.
Range of Values	true | false

datanucleus.persistenceByReachabilityAtCommit

Description	Whether to run the "persistence-by-reachability" algorithm at commit() time. This means that objects that were reachable at a call to makePersistent() but that are no longer persistent will be removed from persistence. For performance improvements, consider turning this off.
Range of Values	true | false

datanucleus.maxFetchDepth

Description	Specifies the default maximum fetch depth to use for fetching operations. The JDO2 specification defines a default of 1, and this is the DataNucleus default, meaning that only the first level of related objects will be fetched by default.
Range of Values	-1 | 1 | positive integer (non-zero)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.classLoaderResolverName

Description	Name of a ClassLoaderResolver to use in class loading. DataNucleus provides a default that implements the JDO2 specification for class loading. This property allows the user to override this with their own class better suited to their own loading requirements.
Range of Values	**jdo** | {name of class-loader-resolver plugin}

datanucleus.primaryClassLoader

Description	Sets a primary classloader for situations where a primary classloader is not accessible. This ClassLoader is used when the class is not found in the default ClassLoader search path. As example, when the database driver is loaded by a different ClassLoader not in the ClassLoader search path for JDO or JPA specifications.
Range of Values	instance of java.lang.ClassLoader

datanucleus.implementationCreatorName

Description	Symbolic name of an implementation creator for "persistent interfaces" (JDO2). DataNucleus provides an implementation creator using ASM. Please note that you should have the DataNucleus Enhancer in the CLASSPATH together with ASM to use "persistent interfaces".
Range of Values	**asm**

datanucleus.plugin.pluginRegistryClassName

Description	Name of a class that acts as registry of plug-ins.
Range of Values	{fully-qualified class name}

datanucleus.plugin.pluginRegistryBundleCheck

Description	Defines what happens when plugin bundles are found and are duplicated
Range of Values	**EXCEPTION** | LOG | NONE

Schema Control

datanucleus.autoCreateSchema

Description	Whether to automatically generate any tables and constraints that don't exist. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.autoCreateTables

Description	Whether to automatically generate any tables that don't exist. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

datanucleus.autoCreateColumns

Description	Whether to automatically generate any columns that don't exist. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

datanucleus.autoCreateConstraints

Description	Whether to automatically generate any constraints that don't exist. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

datanucleus.autoCreateWarnOnError

Description	Whether to only log a warning when errors occur during the auto-creation/validation process. Please use with care since if the schema is incorrect errors will likely come up later and this will postpone those error checks til later, when it may be too late!!
Range of Values	true | false

datanucleus.validateTables

Description	Whether to validate tables against the persistence definition. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

datanucleus.validateColumns

Description	Whether to validate columns against the persistence definition. This refers to the column detail structure and NOT to whether the column exists or not. Please refer to the RDBMS Schema Guide for more details.
Range of Values	true | false

datanucleus.validateConstraints

Description	Whether to validate table constraints against the persistence definition. Please refer to the Schema Guide for more details.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.validateConstraints

Range of Values	true | false

datanucleus.readOnlyDatastore

Description	Whether the datastore is read-only or not (fixed in structure and contents).
Range of Values	true | false

datanucleus.readOnlyDatastoreAction

Description	What happens when a datastore is read-only and an object is attempted to be persisted.
Range of Values	**EXCEPTION** | IGNORE

datanucleus.fixedDatastore

Description	Whether the datastore is fixed in structure or not.
Range of Values	true | false

datanucleus.identifierFactory

Description	Name of the identifier factory to use when generating table/column names etc. See also the JDO RDBMS Identifier Guide.
Range of Values	jpox | jpox2 | jpa | {user-plugin-name}

datanucleus.identifier.case

Description	Which case to use in generated table and column names. See also the JDO RDBMS Identifier Guide.
Range of Values	UpperCase | LowerCase | PreserveCase

datanucleus.identifier.wordSeparator

Description	Separator character(s) to use between words in generated identifiers. Defaults to "_" (underscore)

datanucleus.identifier.tablePrefix

Description	Prefix to be prepended to all generated table names (if the identifier factory supports it)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.identifier.tableSuffix

Description	Suffix to be appended to all generated table names (if the identifier factory supports it)

datanucleus.defaultInheritanceStrategy

Description	How to choose the inheritance strategy default for classes where no strategy has been specified. With *JDO2* this will be "new-table" for base classes and "superclass-table" for subclasses. With *TABLE_PER_CLASS* this will be "new-table" for all classes.
Range of Values	JDO2 | TABLE_PER_CLASS

Transactions and Locking

datanucleus.transactionIsolation

Description	Select the default transaction isolation level for ALL PersistenceManager factories. Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guides for JDO and JPA
Range of Values	read-uncommitted | read-committed | repeatable-read | serializable

datanucleus.SerializeRead

Description	With datastore transactions you can apply locking to objects as they are read from the datastore. This setting applies as the default for all PersistenceManagers/EntityManagers obtained. You can also specify this on a per-transaction or per-query basis (which is often better to avoid deadlocks etc)
Range of Values	true | false

datanucleus.jtaLocator

Description	Selects the locator to use when using JTA transactions so that DataNucleus can find the JTA TransactionManager. If this isn't specified and using JTA transactions DataNucleus will search all available locators which could have a performance impact. See JTA Locator extension. If specifying "custom_jndi" please also specify "datanucleus.jtaJndiLocation"
Range of Values	jboss | jonas | jotm | oc4j | orion | resin | sap | sun | weblogic | websphere | custom_jndi | alias of a JTA transaction locator

datanucleus.jtaJndiLocation

Description	Name of a JNDI location to find the JTA transaction manager from (when using JTA transactions). This is for the case where you know where it is located. If not used DataNucleus will try certain well-known locations

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.jtaJndiLocation

Range of Values	JNDI location

datanucleus.datastoreTransactionDelayOperations

Description	For use when using datastore transactions and has the effect of delaying datastore operations until flush()/commit()
Range of values	true | false

datanucleus.datastoreTransactionFlushLimit

Description	For use when using datastore transactions and is the limit on number of dirty objects before a flush to the datastore will be performed.
Range of values	1 | positive integer

datanucleus.connectionPoolingType

Description	This property allows you to utilise a 3rd party software package for enabling connection pooling using a DataNucleus plugin. Currently DataNucleus supports use of DBCP, C3P0 or Proxool. You must have the plugin and the related 3rd party JARs in your CLASSPATH to use this option. Please refer to the RDBMS Connection Pooling guide for details.
Range of Values	None | DBCP | C3P0 | Proxool

datanucleus.connectionPoolingConfigurationFile

Description	Allows specification of configuration properties for controlling the connection pooling when you have specified the datanucleus.connectionPoolingType above.
Range of Values	Filename present in the CLASSPATH

datanucleus.connection.resourceType

Description	Resource Type for connection ???
Range of Values	JTA | RESOURCE_LOCAL

datanucleus.connection.resourceType2

Description	Resource Type for connection 2
Range of Values	JTA | RESOURCE_LOCAL

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Caching

datanucleus.cache.collections

Description	SCO collections can be used in 2 modes in DataNucleus. You can allow DataNucleus to cache the collections contents, or you can tell DataNucleus to access the datastore for every access of the SCO collection. The default is to use the cached collection.
Range of Values	true | false

datanucleus.cache.collections.lazy

Description	When using cached collections/maps, the elements/keys/values can be loaded when the object is initialised, or can be loaded when accessed (lazy loading). The default is to use lazy loading when the field is not in the current fetch group, and to not use lazy loading when the field is in the current fetch group.
Range of Values	true | false

datanucleus.cache.level1.type

Description	Name of the type of Level 1 cache to use. Defines the backing map.
Range of Values	weak | soft | hard | {your-plugin-name}

datanucleus.cache.level2

Description	Whether to use a Level 2 Cache with this Persistence Manager Factory.
Range of Values	true | false

datanucleus.cache.level2.type

Description	Name of the type of Level 2 Cache to use. Can be used to interface with external caching products.
Range of Values	default | soft | coherence | ehcache | ehcacheclassbased | oscache | swarmcache | {your-plugin-name}

datanucleus.cache.level2.cacheName

Description	Name of the cache. This is for use with plugins such as the Tangosol cache plugin for accessing the particular cache. Please refer to the Cache Guide for JDO or JPA
Range of Values	your cache name

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.cache.level2.configurationFile

Description	The path to the configuration file. e.g. **/cache.xml** The file must be in the classpath and will be looked up as a java resource. Please refer to the Cache Guide for JDO or JPA
Range of Values	your configuration file

Value Generation

datanucleus.valuegeneration.transactionAttribute

Description	Whether to use the PM connection or open a new connection. Only used by value generators that require a connection to the datastore.
Range of Values	New | UsePM

datanucleus.valuegeneration.transactionIsolation

Description	Select the default transaction isolation level for identity generation. Must have *datanucleus.valuegeneration.transactionAttribute* set to *New* Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guides for JDO and JPA
Range of Values	read-uncommitted | read-committed | repeatable-read | serializable

MetaData

datanucleus.metadata.jdoFileExtension

Description	Suffix for JDO MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's.
Range of values	jdo | {file suffix}

datanucleus.metadata.ormFileExtension

Description	Suffix for ORM MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's.
Range of values	orm | {file suffix}

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.metadata.jdoqueryFileExtension

Description	Suffix for JDO Query MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's.
Range of values	jdoquery | {file suffix}

datanucleus.metadata.validate

Description	Whether to validate the MetaData file(s) for XML correctness (against the DTD) when parsing
Range of values	true | false

datanucleus.metadata.autoregistration

Description	Whether to use the JDO auto-registration of metadata. Turned on by default
Range of values	true | false

datanucleus.metadata.supportORM

Description	Whether to support "orm" mapping files. By default we use what the datastore plugin supports. This can be used to turn it off when the datastore supports it but we dont plan on using it (for performance)
Range of values	true | false

Auto-Start

datanucleus.autoStartMechanism

Description	How to initialise DataNucleus at startup. This allows DataNucleus to read in from some source the classes that it was persisting for this data store the previous time. "XML" stores the information in an XML file for this purpose. "SchemaTable" (only for RDBMS) stores a table in the RDBMS for this purpose. "Classes" looks at the property datanucleus.autoStartClassNames for a list of classes. "MetaData" looks at the property datanucleus.autoStartMetaDataFiles for a list of metadata files The other option is "None" (start from scratch each time). Please refer to the Auto-Start Mechanism Guide for more details. The default for RDBMS is "SchemaTable". The default for all other datastores is "None".
Range of Values	XML | Classes | MetaData | None | SchemaTable

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.autoStartMechanismMode

Description	The mode of operation of the auto start mode. Currently there are 3 values. "Quiet" means that at startup if any errors are encountered, they are fixed quietly. "Ignored" means that at startup if any errors are encountered they are just ignored. "Checked" means that at startup if any errors are encountered they are thrown as exceptions.
Range of values	Checked | Ignored | Quiet

datanucleus.autoStartMechanismXmlFile

Description	Filename used for the XML file for AutoStart when using "XML" Auto-Start Mechanism

datanucleus.autoStartClassNames

Description	This property specifies a list of classes (comma-separated) that are loaded at startup when using the "Classes" Auto-Start Mechanism.

datanucleus.autoStartMetaDataFiles

Description	This property specifies a list of metadata files (comma-separated) that are loaded at startup when using the "MetaData" Auto-Start Mechanism.

Query control

datanucleus.query.timeout

Description	The timeout to apply to all queries (millisecs). This also will apply to all fetch statements - for example when retrieving objects using PM.getObjectById().
Range of Values	0 | A positive value (MILLISECONDS)

datanucleus.query.flushBeforeExecution

Description	This property can enforce a flush to the datastore of any outstanding changes just before executing all queries. If using optimistic transactions any updates are typically held back until flush/commit and so the query would otherwise not take them into account.
Range of Values	true | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.query.useFetchPlan

Description	Whether to use the FetchPlan when executing a JDOQL query. The default is to use it which means that the relevant fields of the object will be retrieved. This allows the option of just retrieving the identity columns.
Range of Values	true | false

JPA

datanucleus.jpa.level

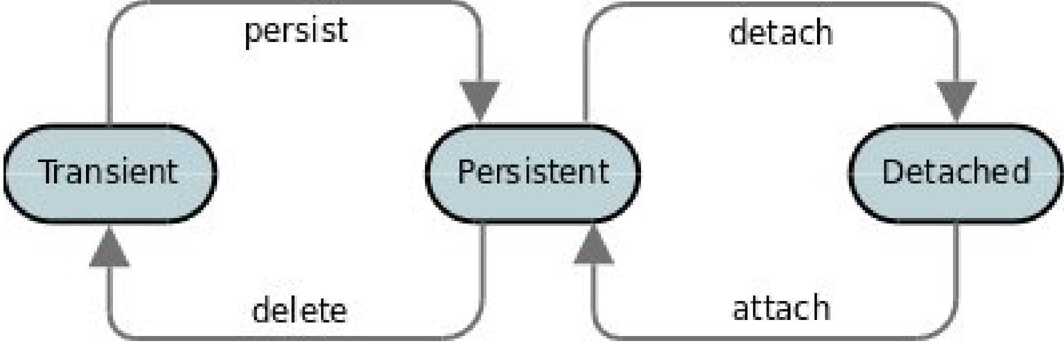
Description	This property defines the level of JPA to be allowed. If you select JPA1 then you get strict JPA1, without DataNucleus extensions and without things like 1-N uni FK relations. If you select JPA2 then you get strict JPA2, without DataNucleus extensions. If you select "DataNucleus" you get full capabilities
Range of Values	**DataNucleus** | JPA1 | JPA2

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.4 Object Lifecycle

Object Lifecycle

During the persistence process, whether using JDO or JPA APIs, an object goes through lifecycle changes. The following diagram highlights the crucial points in the objects lifecycle.



So a newly created object is **transient**. You then persist it (via either JDO *makePersistent* or JPA *persist*) and it becomes **persistent**. You can then detach it for use elsewhere in the application, so it is **detached**. When attaching any changes back to persistence (via JDO *makePersistent* or JPA **merge**) it becomes **persistent** again. Finally when you delete the object from persistence it is in **transient** state.

With JDO there are actually some additional lifecycle states, notably when an object has a field changed, becoming *dirty*, so you get an object in "persistent-dirty", "detached-dirty" states for example.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.5 Performance Tuning

Performance Tuning

DataNucleus, by default, provides certain functionality. In particular circumstances some of this functionality may not be appropriate and it may be desirable to turn on or off particular features to gain more performance for the application in question. This section contains a few common tips

PersistenceManagerFactory/EntityManagerFactory creation

Creation of PersistenceManagerFactory and EntityManagerFactory objects can be expensive and should be kept to a minimum. Depending on the structure of your application, use a single persistence factory per datastore wherever possible. Clearly if your application spans multiple servers then this may be impractical, but should be borne in mind.

You can improve startup speed by setting the property **datanucleus.autoStartMechanism** to *None*. This means that it won't try to load up the classes (or better said the metadata of the classes) handled the previous time that this schema was used. If this isn't an issue for your application then you can make this change. Please refer to the Auto-Start Mechanism for full details.

Some RDBMS (such as Oracle) have trouble returning information across multiple catalogs/schemas and so, when DataNucleus starts up and tries to obtain information about the existing tables, it can take some time. This is easily remedied by specifying the catalog/schema name to be used - either for the PMF as a whole (using the persistence properties **javax.jdo.mapping.Catalog**, **javax.jdo.mapping.Schema**) or for the package/class using attributes in the MetaData. This subsequently reduces the amount of information that the RDBMS needs to search through and so can give significant speed ups when you have many catalogs/schemas being managed by the RDBMS.

Use of PersistenceManager/EntityManager

Clearly the structure of your application will have a major influence on how you utilise a PersistenceManager or EntityManager. A pattern that gives a clean definition of process is to use a different persistence manager for each request to the data access layer. This reduces the risk of conflicts where one thread performs an operation and this impacts on the successful completion of an operation being performed by another thread. Creation of PM/EM's is not an expensive process and use of multiple threads writing to the same persistence manager should be avoided.

Schema Creation

DataNucleus provides 4 PersistenceManagerFactory properties **datanucleus.autoCreateSchema**, **datanucleus.autoCreateTables**, **datanucleus.autoCreateColumns**, and **datanucleus.autoCreateConstraints** that allow creation of the datastore tables. This can cause performance issues at startup. We recommend setting these to *false* at runtime, and instead using SchemaTool to generate any required database schema before running DataNucleus.

Schema Validation

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DataNucleus provides 3 PersistenceManagerFactory properties **datanucleus.validateTables**, **datanucleus.validateConstraints**, **datanucleus.validateColumns** that enforce strict validation of the datastore tables against the Meta-Data defined tables. This can cause performance issues at startup. In general this should be run only at schema generation, and should be turned off for production usage. Set all of these properties to *false*. In addition there is a PMF property **datanucleus.rdbms.CheckExistTablesOrViews** which checks whether the tables/views that the classes map onto are present in the datastore. This should be set to *false* if you require fast start-up. Finally, the property **datanucleus.rdbms.initializeColumnInfo** determines whether the default values for columns are loaded from the database. This property should be set to *NONE* to avoid loading database metadata.

To sum up, the optimal settings with schema creation and validation disabled are:

```
#schema creation
datanucleus.autoCreateSchema=false
datanucleus.autoCreateTables=false
datanucleus.autoCreateColumns=false
datanucleus.autoCreateConstraints=false

#schema validation
datanucleus.validateTables=false
datanucleus.validateConstraints=false
datanucleus.validateColumns=false
datanucleus.rdbms.CheckExistTablesOrViews=false
datanucleus.rdbms.initializeColumnInfo=None
```

O/R Mapping

Where you have an inheritance tree it is best to add a **discriminator** to the base class so that it's simple for DataNucleus to determine the class name for a particular row. This results in cleaner/simpler SQL which is faster to execute. Otherwise it would be necessary to do a UNION of all possible tables.

Database Connection Pooling

DataNucleus, by default, will allocate connections when they are required. It then will close the connection. In addition, when it needs to perform something via JDBC (RDBMS datastores) it will allocate a PreparedStatement, and then discard the statement after use. This can be inefficient relative to a database connection and statement pooling facility such as Apache DBCP. With Apache DBCP a Connection is allocated when required and then when it is closed the Connection isn't actually closed but just saved in a pool for the next request that comes in for a Connection. This saves the time taken to establish a Connection and hence can give performance speed ups the order of maybe 30% or more. You can read about how to enable connection pooling with DataNucleus in the Connection Pooling Guide.

Commit of transaction

DataNucleus verifies if newly persisted objects are memory reachable on commit, if they are not, they are removed from the database. This process mirrors the garbage collection, where objects not referenced are garbage collected or removed from memory. Reachability is expensive because it traverses the whole

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

object tree and may require reloading data from database. If reachability is not needed by your application, you should disable it. To disable reachability set to false the PMF property **datanucleus.persistenceByReachabilityAtCommit**.

DataNucleus will, by default, perform a check on any bidirectional relations to make sure that they are set at both sides at commit. If they aren't set at both sides then they will be made consistent. This check process can involve the (re-)loading of some instances. You can skip this step if you always set *both sides of a relation* by setting the persistence property **datanucleus.manageRelationships** to *false*.

Identity Generators

DataNucleus provides a series of value generators for generation of identity values. These can have an impact on the performance depending on the choice of generator, and also on the configuration of the generator.

- The *max* strategy should not really be used for production since it makes a separate DB call for each insertion of an object. Something like the *increment* strategy should be used instead. Better still would be to choose *native* and let DataNucleus decide for you.
- The *sequence* strategy allows configuration of the datastore sequence. The default can be non-optimum. As a guide, you can try setting **key-cache-size** to 10 and **key-increment-by** to 10.

The native identity generator value is the recommended choice since this will allow DataNucleus to decide which identity generator is best for the RDBMS in use.

Collection/Map caching

DataNucleus has 2 ways of handling calls to SCO Collections/Maps. The original method was to pass all calls through to the datastore. The second method (which is now the default) is to cache the collection/map elements/keys/values. This second method will read the elements/keys/values once only and thereafter use the internally cached values. This second method gives significant performance gains relative to the original method. You can configure the handling of collections/maps as follows :-

- **Globally for the PMF/EMF** - this is controlled by setting the persistence property **datanucleus.cache.collections**. Set it to *true* for caching the collections (default), and *false* to pass through to the datastore.
- **For the specific Collection/Map** - this overrides the global setting and is controlled by adding a MetaData <*collection*> or <*map*> extension **cache**. Set it to *true* to cache the collection data, and *false* to pass through to the datastore.

The second method also allows a finer degree of control. This allows the use of lazy loading of data, hence elements will only be loaded if they are needed. You can configure this as follows :-

- **Globally for the PersistenceManagerFactory** - this is controlled by setting the PMF property **datanucleus.cache.collections.lazy**. Set it to true to use lazy loading, and set it to false to load the elements when the collection/map is initialised.
- **For the specific Collection/Map** - this overrides the global PMF setting and is controlled by adding a MetaData <collection> or <map> extension **cache-lazy-loading**. Set it to true to use lazy

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

loading, and false to load once at initialisation.

NonTransactionRead (Reading persistent objects outside a transaction)

NontransactionalRead has advantages and disadvantages in performance and data freshness in cache. In NontransactionalRead=true mode, the PersistenceManager is able to read objects outside a transaction. The objects read are held cached by the PersistenceManager. The second time a user application requests the same objects from the PersistenceManager they are retrieved from cache. The time spent reading the object from cache is minimum, but the objects may become stale and not represent the database status. If fresh values need to be loaded from the database, then the user application should first call refresh on the object.

Another disadvantage of NontransactionalRead=true mode is due to each operation realized opens a new database connection, but it can be minimized with the use of connection pools.

Reading persistent objects outside a transaction and PersistenceManager

Reading objects outside a transaction and PersistenceManager is a trivial task, but performed in a certain manner can determine the application performance. The objective here is not give you an absolute response on the subject, but point out the benefits and drawbacks for the many possible solutions.

- Use *makeTransient* method.

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.makeTransient(pc);

    pm.currentTransaction().commit();
}
finally
{
    pm.close();
}
//read the persistent object here
System.out.prinln(pc.getName());
```

- Use RetainValues=true.

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().setRetainValues(true);
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    pm.currentTransaction().commit();
}
finally
{
    pm.close();
}
//read the persistent object here
System.out.prinln(pc.getName());
```

- Use *detachCopy* method.

```
Object copy = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    Object pc = pm.getObjectById(id);
    copy = pm.detachCopy(pc);

    pm.currentTransaction().commit();
}
finally
{
    pm.close();
}
//read or change the detached object here
System.out.prinln(copy.getName());
```

- Use *detachAllOnCommit*.

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.setDetachAllOnCommit(true);
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.currentTransaction().commit(); // Object "pc" is now detached
}
finally
{
    pm.close();
}
//read or change the detached object here
System.out.prinln(pc.getName());
```

The most expensive in terms of performance is the *detachCopy* because it makes copies of persistent objects. The advantage of detachment (via *detachCopy* or *detachAllOnCommit*) is that changes made outside

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

the transaction can be futher used to update the database in a new transaction. The other methods also allow changes outside of the transaction, but the changed instances can't be used to update the database.

In *RetainValues=true* and *makeTransient* no object copies are made and the object values are set down in instances when the PersistenceManager disassociates them. Both methods are equivalent in performance, however the *makeTransient* method will set the values of the object during the instant the *makeTransient* method is invoked, and the *RetainValues=true* will set values of the object during commit.

The bottom line is to not use detachment if instances will only be used to read values.

Logging

I/O consumes a huge slice of the total processing time. Therefore it is recommended to reduce or disable logging in production. To disable the logging set the DataNucleus category to OFF in the Log4j configuration. See Logging for more information.

```
log4j.category.DataNucleus=OFF
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.6 Failover

Failover

In the majority of production situations it is desirable to have a level of failover between the underlying datastores used for persistence. You have at least 2 options available to you here. These are shown below

Sequoia

Sequoia is a transparent middleware solution offering clustering, load balancing and failover services for any database. Sequoia is the continuation of the C-JDBC project. The database is distributed and replicated among several nodes and Sequoia balances the queries among these nodes. Sequoia handles node and network failures with transparent failover. It also provides support for hot recovery, online maintenance operations and online upgrades.

Sequoia can be used with DataNucleus by just providing the Sequoia datastore URLs as input to DataNucleus. **There is a problem outstanding in Sequoia itself in that its JDBC driver doesnt provide DataNucleus with the correct major/minor versions of the underlying datastore. Until Sequoia fix this issue, use of Sequoia will be unreliable**

DataNucleus Failover capability

DataNucleus has the capability to switch to between DataSources upon failure of one while obtaining a datastore connection. The failover mechanism is useful for applications with multiple database nodes when the data is actually replicated/synchronized by the underlying database. There are 2 things to be aware of before utilising this functionality.

- DataNucleus doesn't replicate changes to all database nodes, and for this reason, this feature is suggested to be used only for reading objects or if the database is capable to replicate the changes to all nodes.
- If a connection breaks while in use the failover mechanism will not handle it, thus the user application must take care of restarting the transaction and execute the operations.

Several failover algorithm are allowed to be used, one at time, as for example *round-robin*, *ordered list* or *random*. The default algorithm, ordered list, is described below and is provided by DataNucleus. You can also implement and plug your own algorithm. See Connection Provider.

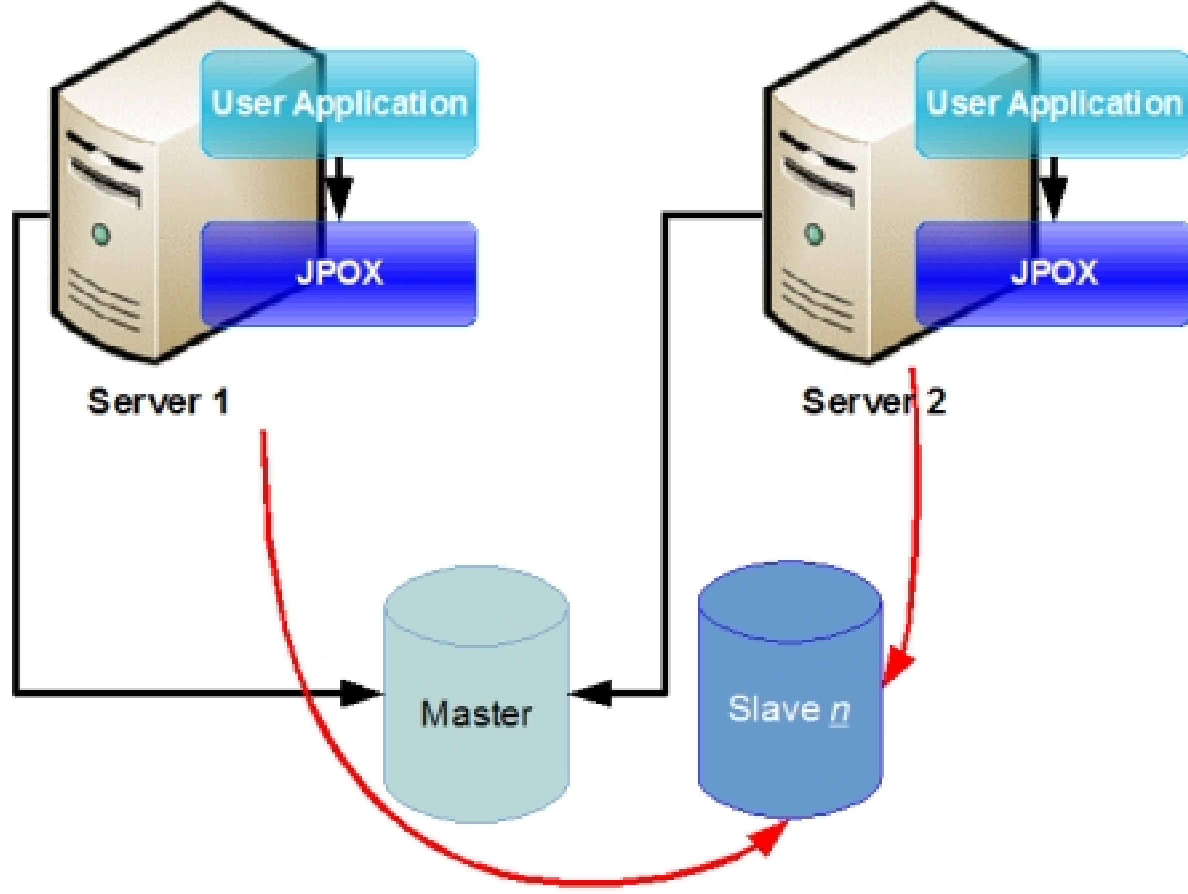
To use failover, each datastore connection must be provided through DataSources. The *datanucleus.ConnectionFactoryName* property must be declared with a list of JNDI names pointing to DataSources, in the form of *<JNDINAME> [,<JNDINAME>]*. See the example:

```
datanucleus.ConnectionFactoryName=JNDINAME1,JNDINAME2
```

ALL RIGHTS RESERVED

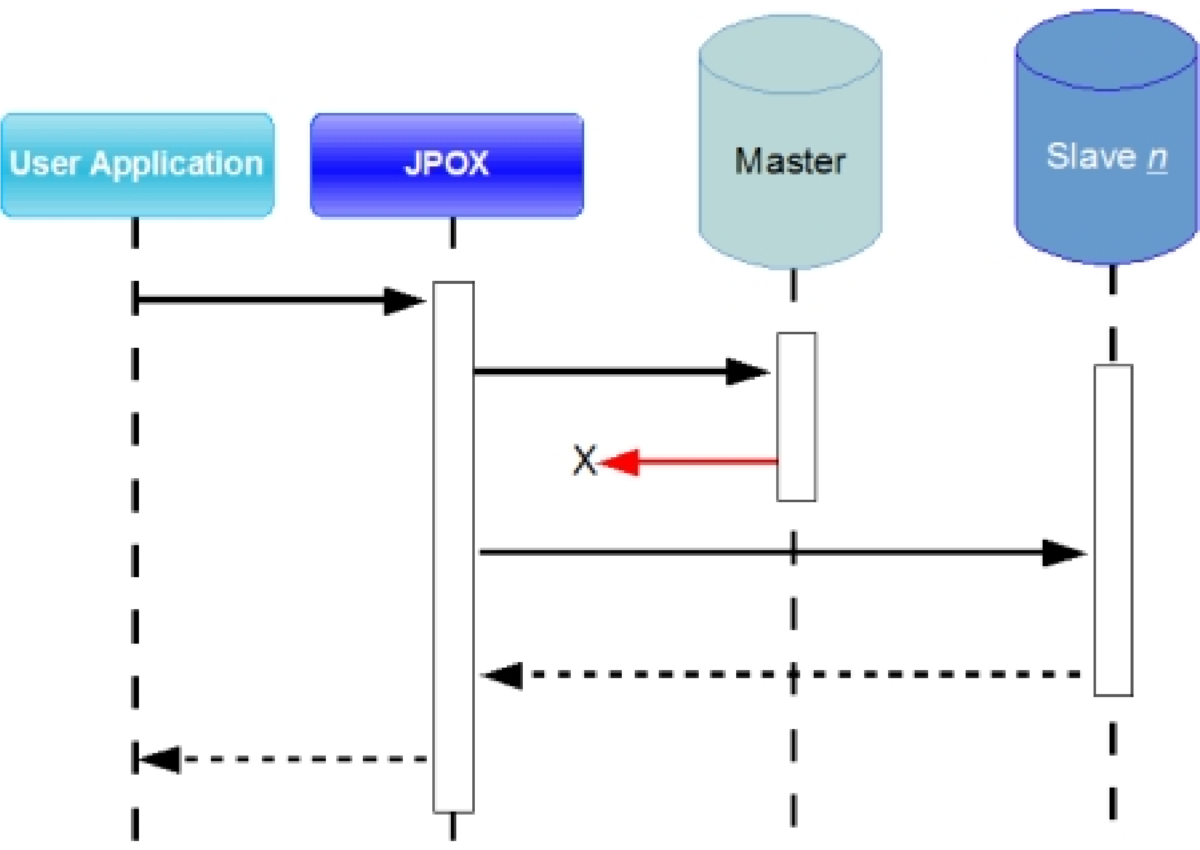
At least one least one JNDI name must be declared.

The **Ordered List Algorithm (default)** allows you to switch to slave DataSources upon failure of a master DataSource while obtaining a datastore connection. This is shown below.



Each time DataNucleus needs to obtain a connection to the datastore, it takes the first DataSource, the Master, and tries, on failure to obtain the connection goes to the next on the list until it obtains a connection to the datastore or the end of the list is reached.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The first JNDI name in the *datanucleus.ConnectionFactoryName* property is the Master DataSource and the following JNDI names are the Slave DataSources.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.7 Security

Java Security Manager

The Java Security Manager can be used with DataNucleus to provide a security platform to sensitive applications.

To use the Security Manager, specify the *java.security.manager* and *java.security.policy* arguments when starting the JVM. e.g.

```
java -Djava.security.manager
-Djava.security.policy==/etc/apps/security/security.policy ...
```

Note that when you use *-Djava.security.policy*==... (double equals sign) you override the default JVM security policy files, while if you use *-Djava.security.policy*=... (single equals sign), you append the security policy file to any existing ones.

The following is a sample security policy file to be used with DataNucleus.

```
grant codeBase "file:${/}jdo2-api-2.0.jar" {

    //jdo API needs datetime (timezone class needs the following)
    permission java.util.PropertyPermission "user.country", "read";
    permission java.util.PropertyPermission "user.variant", "read";
    permission java.util.PropertyPermission "user.timezone", "read,write";
    permission java.util.PropertyPermission "java.home", "read";
};
grant codeBase "file:${/}datanucleus*.jar" {

     //jdo
    permission javax.jdo.spi.JDOPermission "getMetadata";
    permission javax.jdo.spi.JDOPermission "setStateManager";

    //DataNucleus needs to get classloader of classes
    permission java.lang.RuntimePermission "getClassLoader";

    //DataNucleus needs to detect the java and os version
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "os.name", "read";

    //DataNucleus reads these system properties
    permission java.util.PropertyPermission "datanucleus.*", "read";
    permission java.util.PropertyPermission "javax.jdo.*", "read";

    //DataNucleus runtime enhancement (needs read access to all jars/classes in
classpath,
    // so use <<ALL FILES>> to facilitate config)
    permission java.lang.RuntimePermission "createClassLoader";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    //DataNucleus needs to read manifest files (read permission to location of
MANIFEST.MF files)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    permission java.io.FilePermission "${user.dir}${/}-", "read";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    //DataNucleus uses reflection!!!
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.8 Troubleshooting

Troubleshooting

This section describes the most common problems found when using DataNucleus in different architectures. It describes symptoms and methods for collecting data for troubleshooting thus reducing time to narrow the problem down and come to a solution.

Out Of Memory error

Introdution

Java allocate objects in the runtime memory data area called *heap*. The heap is created on virtual machine start-up. The memory allocated to objects are reclaimed by Garbage Collectors when the object is no longer referenced (See Object References). The heap may be of a fixed size, but can also be expanded when more memory is needed or contracted when no longer needed. If a larger heap is needed and it cannot be allocated an *OutOfMemory* is thrown. See JVM Specification.

Native memory is used by the JVM to perform its operations like creation of threads, sockets, jdbc drivers using native code, libraries using native code, etc.

The maximum size of heap memory is determined by the -Xmx on the java command line. If Xmx is not set, then the JVM decides for the maximum heap. The heap and native memory are limited to the maximum memory allocated by the JVM. For example, if the JVM Xmx is set to 1GB and currently use of native memory is 256MB then the heap can only use 768MB.

Causes

Common causes of out of memory:

- Not enough heap - The JVM needs more memory to deal with the application requirements. Queries returning more objects than usual can be the cause.
- Not enough PermGen - The JVM needs more memory to load class definitions.
- Memory Leaks - The application does not close the resources, like the PersistenceManager, EntityManager or Queries, and the JVM cannot reclaim the memory.
- Caching - Caching in the application or inside DataNucleus holding strong references to objects.
- Garbage Collection - If no full garbage collection is performed before the OutOfMemory it can indicate a bug in the JVM Garbage Collector.
- Memory Fragmentation - A large object needs to be placed in the memory, but the JVM cannot allocate a continous space to it because the memory is fragmented.
- JDBC driver - a bug in the JDBC driver not flushing resources or keeping large result sets in memory.

Throubleshooting

JVM

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Collect garbage collection information by adding *-verbosegc* to the java command line. The *verbosegc* flag will print garbage collections to System output.

Sun JVM

The Sun JVM 1.4 or upper accepts the flag *-XX:+PrintGCDetails*, which prints detailed information on Garbage Collections.

The Sun JVM accepts the flag *-verbose:class*, which prints information about each class loaded. This is useful to troubleshoot issues when OutOfMemory occurs due to lack of space in the PermGen, or when NoClassDefFoundError or Linkage errors occurs.

The Sun JVM 1.5 or upper accepts the flag *-XX:+HeapDumpOnOutOfMemoryError*, which creates a hprof binary file head dump in case of an OutOfMemoryError. You can analyse the heap dump using tools such as jhat or YourKit profiler.

DataNucleus

DataNucleus keeps in cache persistent objects using weak references by default. Enable debug mode DataNucleus.Cache category to investigate the size of the cache in DataNucleus.

Resolution

DataNucleus can be configured to reduce the number of objects in cache. DataNucleus has cache for persistent objects, metadata, datastore metadata, fields of type Collection or Map, or query results.

Query Results Cache

The query results hold strong references to the retrieved objects. If a query returns too many objects it can lead to OutOfMemory error. To be able to query over large result sets, change the result set type to *scroll-insensitive* in the pmf setting *datanucleus.rdbms.query.resultSetType*.

Query leak

The query results are kept in memory until the PersistenceManager or Query are closed. To avoid memory leaks caused by queries in memory, it's capital to explicitly close the query as soon as possible. The following snippet shows how to do it.

```
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE
price < :limit");
List results = (List)query.execute(new Double(200.0));
//...
//...
//closes the query
query.closeAll();
```

PersistenceManager leak

It's also a best practice to ensure the PersistenceManager is closed in a try finally block. The PersistenceManager/EntityManager has level 1 cache of persistence objects. See the following example:

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    //...
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Cache for fields of Collection or Map

If collection or map fields have large number of elements, the caching of elements can be disabled with the pmf *datanucleus.cache.collections* setting it to false.

Persistent Objects cache

The cache control of persistent objects is described in the Cache Guide for JDO or JPA

Metadata and Datastore Metadata cache

The metadata and datastore metadata caching cannot be controled by the application, because the memory required for it is insignificant.

OutOfMemory when persisting new objects

When persistent many objects, the flush operation should be periodically invoked. This will give a hint to DataNucleus to flush the changes to the database and release the memory. In the below sample the *pm.flush()* operation is invoked on every 10,000 objects persisted.

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    for (int i=0; i<100000; i++)
    {
        Wardrobe wardrobe = new Wardrobe();
        wardrobe.setModel("3 doors");
        pm.makePersistent(wardrobe);
        if (i % 10000 == 0)
        {
            pm.flush();
        }
    }
    tx.commit();
}
finally
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Frozen application

Introdution

The application pauses for short or long periods or hangs during very long time.

Causes

Common causes:

- Database Locking - Database waiting other transactions to release locks due to deadlock or locking contentions.
- Garbage Collection Pauses - The garbage collection pauses the application to free memory resources.
- Application Locking - Thread 2 waiting for resources locked by Thread 1.

Throubleshooting

Database locking

Use a database specific tool or database scripts to find the current database locks.

In Microsoft SQL, the stored procedured *sp_lock* can be used to examinate the database locks.

Query Timeout

To avoid database locking to hang the application when a query is performed, set the query timeout. See Query Timeout.

Garbage Collection pauses

Check if the application freezes when the garbage collection starts. Add *-verbosegc* to the java command line and restart the application.

Application Locking

Thread dumps are snapshots of the threads and monitors in the JVM. Thread dumps help to diagnose applications by showing what the application is doing at a certain moment of time.

To generate Thread Dumps in MS Windows, press <ctrl><break> in the window running the java application.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

To generate Thread Dumps in Linux/Unix, execute kill -3 *process_id*

To effectivelly diagnose the problem, take 5 Thread Dumps with 3 to 5 seconds internal between each one.

See An Introduction to Java Stack Traces.

Postgres

ERROR: schema does not exist

Problem

Exception *org.postgresql.util.PSQLException: ERROR: schema "PUBLIC" does not exist* raised during transaction.

Troubleshooting

- Verify that the schema "PUBLIC" exists. If the name is lowercased ("public"), set *datanucleus.identifier.case=PreserveCase*, since Postgres is case sensitive.
- Via pgAdmin Postgres tool, open a connection to the schema and verify it is acessible with issuing a *SELECT 1* statement.

Command Line Tools

CreateProcess error=87

Problem

CreateProcess error=87 when running DataNucleus tools under Microsoft Windows OS.

Windows has a command line length limitation, between 8K and 64K characters depending on the Windows version, that may be triggered when running tools such as the Enhancer or the SchemaTool with too many arguments.

Solution

When running such tools from Maven or Ant, disable the fork mechanism by setting the option fork="false".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.9 Management (JMX)

Management (JMX) with JVM MBean Server

DataNucleus provides MBeans that can be used to monitor, manage and configure DataNucleus at runtime. More about JMX here.

An MBean server is bundled with Sun JRE since its version 1.5, and you can easily activate DataNucleus MBeans registration by adding the DataNucleus Management Platform plugin found in jar *datanucleus-management-{version}.jar* to the classpath, and creating your PMF/EMF with the persistence property **datanucleus.managedRuntime** as true

Additionaly, setting a few system properties are necessary for configuring the Sun JMX implementation. The minimum properties required are the following:

- com.sun.management.jmxremote
- com.sun.management.jmxremote.authenticate
- com.sun.management.jmxremote.ssl
- com.sun.management.jmxremote.port=<port number>

Usage example:

```
java -cp TheClassPathInHere
     -Dcom.sun.management.jmxremote
     -Dcom.sun.management.jmxremote.authenticate=false
     -Dcom.sun.management.jmxremote.ssl=false
     -Dcom.sun.management.jmxremote.port=8001
     TheMainClassInHere
```

Once you start your application and DataNucleus is initialized you can browse DataNucleus MBeans using a tool called jconsole (jconsole is distributed with the Sun JDK) via the URL:

```
service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi
```

Note that the mode of usage is presented in this document as matter of example, and by no means we recommend to disable authentication and secured communication channels. Further details on the Sun JMX implementation and how to configure it properly can be found in here.

DataNucleus MBeans are registered in a MBean Server when DataNucleus is started up (e.g. upon JDO PMF or JPA EMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

Management (JMX) with MX4J

DataNucleus provides MBeans that can be used to monitor, manage and configure DataNucleus at runtime. More about JMX here. To enable management using MX4J you must

- specify the persistence property **datanucleus.managedRuntime** as true when creating the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

PMF/EMF

- have the **datanucleus-mx4j** plugin jar in the CLASSPATH.

DataNucleus MBeans are registered in a MBean Server when DataNucleus is started up (e.g. upon JDO PMF or JPA EMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.10 Logging

DataNucleus Logging

DataNucleus can be configured to log significant amounts of information regarding its process. This information can be very useful in tracking the persistence process, and particularly if you have problems. DataNucleus will log as follows :-

- **Log4J** - if you have Log4J in the CLASSPATH, Apache Log4J will be used
- **JDK1.4** - if you don't have Log4J in the CLASSPATH, but you are using JDK1.4 or later *java.util.logging* will be used
- **No logging** - if you have neither Log4J, nor JDK1.4+ then no logging will be performed

DataNucleus logs messages to various categories (in Log4J and JDK1.4 these correspond to a "Logger"), allowing you to filter the logged messages by these categories - so if you are only interested in a particular category you can effectively turn the others off. DataNucleus's log is written by default in English. If your JDK is running in a Spanish locale then your log will be written in Spanish. If you have time to translate our log messages into other languages, please contact one of the developers via the Online Forum.

Logging Categories

DataNucleus uses a series of categories, and logs all messages to these categories. Currently DataNucleus uses the following

- **DataNucleus.JDO** - All messages general to JDO
- **DataNucleus.JPA** - All messages general to JPA
- **DataNucleus.Persistence** - All messages relating to the persistence process
- **DataNucleus.Query** - All messages relating to queries
- **DataNucleus.Lifecycle** - All messages relating to object lifecycle changes
- **DataNucleus.Reachability** - All messages relating to "persistence-by-reachability"
- **DataNucleus.Cache** - All messages relating to the DataNucleus Cache
- **DataNucleus.ClassLoading** - All exceptions relating to class loading issues
- **DataNucleus.MetaData** - All messages relating to MetaData
- **DataNucleus.Management** - All messages relating to Management
- **DataNucleus.General** - All general operational messages
- **DataNucleus.Connection** - All messages relating to Connections.
- **DataNucleus.JCA** - All messages relating to Connector JCA.
- **DataNucleus.Transaction** - All messages relating to transactions
- **DataNucleus.Plugin** - All messages relating to DataNucleus plug-ins
- **DataNucleus.ValueGeneration** - All messages relating to value generation
- **DataNucleus.Datastore** - All general datastore messages
- **DataNucleus.Datastore.Schema** - All schema related datastore log messages

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- **DataNucleus.Datastore.Persist** - All datastore persistence messages
- **DataNucleus.Datastore.Retrieve** - All datastore retrieval messages
- **DataNucleus.Datastore.Native** - Log of all 'native' statements sent to the datastore
- **DataNucleus.Enhancer** - All messages from the DataNucleus Enhancer.
- **DataNucleus.SchemaTool** - All messages from DataNucleus SchemaTool
- **DataNucleus.IDE** - Messages from the DataNucleus IDE.

Using Log4J

Log4J allows logging messages at various severity levels. The levels used by Log4J, and by DataNucleus's use of Log4J are DEBUG, INFO, WARN, ERROR, FATAL. Each message is logged at a particular level to a category (as described above). The other setting is OFF which turns off a logging category. This is very useful in a production situation where maximum performance is required.

To enable the DataNucleus log, you need to provide a Log4J configuration file when starting up your application. This may be done for you if you are running within a J2EE application server (check your manual for details). If you are starting your application yourself, you would set a JVM parameter as

```
-Dlog4j.configuration=file:log4j.properties
```

where log4j.properties is the name of your Log4J configuration file. Please note the "file:" prefix to the file since a URL is expected. [When using JDK 1.4 logging you need to specify the system property "java.util.logging.config.file"]

The Log4J configuration file is very simple in nature, and you typically define where the log goes to (e.g to a file), and which logging level messages you want to see. Here's an example

```
# Define the destination and format of our logging
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.File=datanucleus.log
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{HH:mm:ss,SSS} (%t) %-5p [%c] - %m%n

# DataNucleus Categories
log4j.category.DataNucleus.JDO=INFO, A1
log4j.category.DataNucleus.Cache=INFO, A1
log4j.category.DataNucleus.MetaData=INFO, A1
log4j.category.DataNucleus.General=INFO, A1
log4j.category.DataNucleus.Utility=INFO, A1
log4j.category.DataNucleus.Transaction=INFO, A1
log4j.category.DataNucleus.Datastore=DEBUG, A1
log4j.category.DataNucleus.ClassLoading=DEBUG, A1
log4j.category.DataNucleus.Plugin=DEBUG, A1
log4j.category.DataNucleus.ValueGeneration=DEBUG, A1

log4j.category.DataNucleus.Enhancer=INFO, A1
log4j.category.DataNucleus.SchemaTool=INFO, A1
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

In this example, I am directing my log to a file (*datanucleus.log*). I have defined a particular "pattern" for the messages that appear in the log (to contain the date, level, category, and the message itself). In addition I have assigned a level "threshold" for each of the DataNucleus categories. So in this case I want to see all messages down to DEBUG level for the DataNucleus RDBMS persister.

Performance Tip : Turning OFF the logging, or at least down to ERROR level provides a *significant* improvement in performance. With Log4J you do this via

```
log4j.category.DataNucleus=OFF
```

Using java.util.logging

java.util.logging allows logging messages at various severity levels. The levels used by java.util.logging, and by DataNucleus's internally are fine, info, warn, severe. Each message is logged at a particular level to a category (as described above).

By default, the *java.util.logging* configuration is taken from a properties file <JRE_DIRECTORY>/lib/logging.properties". Modify this file and configure the categories to be logged, or use the *java.util.logging.config.file* system property to specify a properties file (in java.util.Properties format) where the logging configuration will be read from. Here is an example:

```
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
DataNucleus.General.level=fine
DataNucleus.JDO.level=fine

# --- ConsoleHandler ---
# Override of global logging level
java.util.logging.ConsoleHandler.level=SEVERE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# --- FileHandler ---
# Override of global logging level
java.util.logging.FileHandler.level=SEVERE

# Naming style for the output file:
java.util.logging.FileHandler.pattern=datanucleus.log

# Limiting size of output file in bytes:
java.util.logging.FileHandler.limit=50000

# Number of output files to cycle through, by appending an
# integer to the base file name:
java.util.logging.FileHandler.count=1

# Style of output (Simple or XML):
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

Please read the javadocs for *java.util.logging* for additional details on its configuration.

Sample Log Output

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here is a sample of the type of information you may see in the DataNucleus log when using Log4J.

```
21:26:09,000 (main) INFO  DataNucleus.Datastore.Schema - Adapter initialised :
MySQLAdapter, MySQL version 4.0.11
21:26:09,365 (main) INFO  DataNucleus.Datastore.Schema - Creating table
null.DELETE_ME1080077169045
21:26:09,370 (main) DEBUG DataNucleus.Datastore.Schema - CREATE TABLE
DELETE_ME1080077169045
(
    UNUSED INTEGER NOT NULL
) TYPE=INNODB
21:26:09,375 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 3 ms
21:26:09,388 (main) WARN  DataNucleus.Datastore.Schema - Schema Name could not be
determined for this datastore
21:26:09,388 (main) INFO  DataNucleus.Datastore.Schema - Dropping table
null.DELETE_ME1080077169045
21:26:09,388 (main) DEBUG DataNucleus.Datastore.Schema - DROP TABLE
DELETE_ME1080077169045
21:26:09,392 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 3 ms
21:26:09,392 (main) INFO  DataNucleus.Datastore.Schema - Initialising Schema ""
using "SchemaTable" auto-start
21:26:09,401 (main) DEBUG DataNucleus.Datastore.Schema - Retrieving type for table
DataNucleus_TABLES
21:26:09,406 (main) INFO  DataNucleus.Datastore.Schema - Creating table
null.DataNucleus_TABLES
21:26:09,406 (main) DEBUG DataNucleus.Datastore.Schema - CREATE TABLE
DataNucleus_TABLES
(
    CLASS_NAME VARCHAR (128) NOT NULL UNIQUE ,
    `TABLE_NAME` VARCHAR (127) NOT NULL UNIQUE
) TYPE=INNODB
21:26:09,416 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 10 ms
21:26:09,417 (main) DEBUG DataNucleus.Datastore - Retrieving type for table
DataNucleus_TABLES
21:26:09,418 (main) DEBUG DataNucleus.Datastore - Validating table :
null.DataNucleus_TABLES
21:26:09,425 (main) DEBUG DataNucleus.Datastore - Execution Time = 7 ms
```

So you see the time of the log message, the level of the message (DEBUG, INFO, etc), the category (DataNucleus.Datastore, etc), and the message itself. So, for example, if I had set the *DataNucleus.Datastore.Schema* to DEBUG and all other categories to INFO I would see *all* DDL statements sent to the database and very little else.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

2.11 ORM Relationships

ORM Relationships

There are 2 prevalent persistence specifications in the Java ORM world. JDO2 provides the most complete definition, whilst JPA is the most recent. In this guide we show the different types of ORM relation commonly used, and mark against it which specification supports it. This list is not yet complete but will be added to to provide a comprehensive list of relationship type and where you can find it.

Field Type	Relation	JDO2	JPA1	DataNucleus
PC	1-1 Unidirectional	✓	✓	✓
PC	1-1 Bidirectional	✓	✓	✓
PC	1-1 serialised	✓	✓	✓
PC	1-1 CompoundIdentity Unidirectional	✓	✗	✓
PC	1-N CompoundIdentity Bidirectional	✓	✗	✓
Interface	1-1 Unidirectional	✓	✗	✓
Interface	1-1 Bidirectional	✓	✗	✓
Interface	1-1 serialised	✓	?	✓
Collection<PC>	1-N ForeignKey Unidirectional Collection	✓	✗	✓
Collection<PC>	1-N ForeignKey Bidirectional Collection	✓	✓	✓
Collection<PC>	1-N JoinTable Unidirectional Collection	✓	✓	✓
Collection<PC>	1-N JoinTable Bidirectional Collection	✓	✓	✓
Collection<Non-PC>	1-N JoinTable Collection	✓	✗	✓
Collection<PC>	1-N JoinTable Collection using shared JoinTable	✗	✗	✓
Collection<PC>	1-N ForeignKey Collection using shared ForeignKey	✗	✗	✓
Collection<PC>	M-N JoinTable	✓	✓	✓
Collection<PC>	1-N CompoundIdentity Unidirectional	✓	✗	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Field Type	Relation	JDO2	JPA1	DataNucleus
Collection<PC>	1-N serialised Collection	✓	✓	✓
Collection<PC>	1-N JoinTable Collection of serialised elements	✓	✗	✓
List<PC>	1-N ForeignKey Unidirectional Indexed List	✓	✗	✓
List<PC>	1-N ForeignKey Bidirectional Indexed List	✓	✗	✓
List<PC>	1-N JoinTable Unidirectional Indexed List	✓	✗	✓
List<PC>	1-N JoinTable Bidirectional Indexed List	✓	✗	✓
List<Non-PC>	1-N JoinTable Indexed List	✓	✗	✓
List<PC>	1-N ForeignKey Unidirectional Ordered List	✗	✓	✓
List<PC>	1-N ForeignKey Bidirectional Ordered List	✗	✓	✓
List<PC>	1-N JoinTable Unidirectional Ordered List	✗	✓	✓
List<PC>	1-N JoinTable Bidirectional Ordered List	✗	✓	✓
Map<PC, PC>	1-N JoinTable Map	✓	✗	✓
Map<Non-PC, PC>	1-N JoinTable Map	✓	✗	✓
Map<PC, Non-PC>	1-N JoinTable Map	✓	✗	✓
Map<Non-PC, Non-PC>	1-N JoinTable Map	✓	✗	✓
Map<Non-PC, PC>	1-N ForeignKey Map Unidirectional (key stored in value)	✓	✓	✓
Map<Non-PC, PC>	1-N ForeignKey Map Bidirectional (key stored in value)	✓	✓	✓
Map<PC, Non-PC>	1-N ForeignKey Map Unidirectional (value stored in key)	✓	✗	✓
Map<PC, PC>	1-N serialised Map	✓	✓	✓
Map<PC, PC>	1-N JoinTable Map of serialised keys/values	✓	✗	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Field Type	Relation	JDO2	JPA1	DataNucleus
PC[ ]	1-N ForeignKey Unidirectional Array	✓	✗	✓
PC[ ]	1-N JoinTable Unidirectional Array	✓	✗	✓
PC[ ]	1-N serialised Array	✓	✓	✓
Non-PC[ ]	1-N JoinTable Unidirectional Array	✓	✗	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.1 JDO Class Mapping

Class Mapping

When persisting a class you need to decide how it is to be mapped to the datastore. By this we mean which fields of the class are persisted. DataNucleus knows how to persist certain Java types and so you bear this list in mind when deciding which fields to persist. Also please note that JDO **cannot persist static or final fields**. Let's take a sample class as an example

```
public class Hotel
{
    private long id; // identity
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    private String hotelNickname;
    private Set rooms = new HashSet();
    private Manager manager;
    ...
}
```

We have a series of fields and we want to persist all fields apart from *hotelNickname* which is of no real use in our system. In addition, we want our *Hotel* class to be detachable, meaning that we can detach objects of that type update them in a different part of our system, and the attach them again.

We can define this basic persistence information in 3 ways - with XML MetaData, with JDK1.5 Annotations or with a mix of MetaData and Annotations. We show all ways here.

MetaData

To achieve the above aim we define our Meta-Data like this

```
<class name="Hotel" detachable="true">
    <field name="id" primary-key="true"/>
    <field name="name"/>
    <field name="address"/>
    <field name="telephoneNumber"/>
    <field name="numberOfRooms"/>
    <field name="hotelNickname" persistence-modifier="none"/>
    <field name="rooms">
        <collection element-type="Room"/>
    </field>
    <field name="manager"/>
</class>
```

Note the following

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- We have identified the *id* field as the primary key. We didn't bother specifying a Primary Key class since there is only a single PK field. By doing this we have selected application-identity
- We have included all fields in the MetaData, although if you look at the Types Guide you see the column "Persistent?". All "simple" types like String, int are by default persistent and so we could have omitted these since they are by default going to be persisted.
- We have used *persistence-modifier* for the *hotelNickname* field to make it non-persistent.
- Our Set field we have identified the type of the element that it contains. This is compulsory for collection and map fields. If the field is declared using JDK1.5 generics then you can safely omit this since all necessary information is in the class declaration.
- We have added the attribute "detachable" as true for the class.

So it is really very simple. This first step is to define the basic persistence of a class. If you are using a datastore (such as RDBMS) that requires detailed mapping information then you now need to proceed to the Schema Mapping Guide. If however you are using a datastore that doesnt need such information (such as DB4O) then you have defined the persistence of your class.

See also :-

- MetaData reference for <class> element
- MetaData reference for <field> element
- MetaData reference for <collection> element
- MetaData reference for <map> element

Annotations

Here we are using JDK1.5 or higher and we annotate the class directly using JDO Annotations We annotate the class like this

```
@PersistenceCapable
public class Hotel
{
    @Persistent(primaryKey="true")
    private long id;

    @Persistent
    private String name;

    @Persistent
    private String address;

    @Persistent
    private String telephoneNumber;

    @Persistent
    private int numberOfRooms;

    @Persistent(persistenceModifier=PersistenceModifier.NONE)
    private String hotelNickname;

    @Persistent
    @Element(types=org.datanucleus.samples.Room.class)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    private Set rooms = new HashSet();

    @Persistent
    private Manager manager;
    ...
}
```

Note that we could have omitted the @Element if we had declared the *rooms* field as Set<Room>.

See also :-

- Annotations reference for @PersistenceCapable
- Annotations reference for @PersistenceAware
- Annotations reference for @Persistent
- Annotations reference for @Transient
- Annotations reference for @Transactional
- Annotations reference for @Element
- Annotations reference for @Key
- Annotations reference for @Value

Annotations + MetaData

If we are using JDK 1.5+ we can take advantage of Annotations, but we want to take into account the disadvantage of Annotations, namely that we may want to deploy our application to multiple datastores. This means that we would be extremely unwise to specify ORM information in Annotations. With this in mind we decide that we will specify just the basic persistence information (which classes/fields are persisted etc) using Annotations, and the remainder will go in MetaData.

The order of precedence for persistence information is

- ORM MetaData definition
- JDO MetaData definition
- Annotations definition

So anything specified in MetaData will override all Annotations.

Persistence Aware

With JDO persistence all classes that are persisted have to be identified in MetaData or annotations as shown above. In addition, if any of your other classes **access the fields of these persistable classes directly** then these other classes should be defined as *PersistenceAware*. You do this as follows

```
<class name="MyClass" persistence-modifier="persistence-aware"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

or with annotations

```
@PersistenceAware
public class MyClass
{
    ...
}
```

See also :-

- Annotations reference for @PersistenceAware

Overriding Superclass MetaData

If you are using XML MetaData you can also override the MetaData for fields/properties of superclasses. You do this by adding an entry for {*class-name*}.*fieldName*, like this

```
<class name="Hotel" detachable="true">
    ...
    <field name="HotelSuperclass.someField" default-fetch-group="false"/>
```

so we have changed the field "someField" specified in the persistent superclass "HotelSuperclass" to not be part of the DFG.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.2 Java Types

JDO : Persistable Java Types

When persisting a class, a persistence solution needs to know how to persist the types of each field in the class. Clearly a persistence solution can only support a finite number of Java types; it cannot know how to persist every possible type creatable. The JDO specification define lists of types that are required to be supported by all implementations of those specifications. This support can be conveniently split into two parts

First-Class (FCO) Types

An object that can be *referred to* (object reference, providing a relation) and that has an "identity" is termed a **First Class Object (FCO)**. DataNucleus supports the following Java types as FCO

- **PersistenceCapable** : any class marked for persistence can be persisted with its own identity in the datastore
- **interface** where the field represents a *PersistenceCapable* object
- **java.lang.Object** where the field represents a *PersistenceCapable* object

Supported Second-Class (SCO) Types

An object that does not have an "identity" is termed a **Second Class Object (SCO)**. This is something like a String or Date field in a class, or alternatively a Collection (that contains other objects). The table below shows the currently supported SCO java types in DataNucleus. The table shows

- **Extension?** : whether the type is JDO2 standard, or is a DataNucleus extension
- **default-fetch-group (DFG)** : whether the field is retrieved by default when retrieving the object itself
- **persistence-modifier** : whether the field is persisted by default, or whether the user has to mark the field as persistent in XML/annotations to persist it
- **proxied** : whether the field is represented by a "proxy" that intercepts any operations to detect whether it has changed internally.
- **primary-key** : whether the field can be used as part of the primary-key

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
boolean		✔	✔	✘	✔
byte		✔	✔	✘	✔
char		✔	✔	✘	✔
double		✔	✔	✘	✘

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
float		✓	✓	✗	✗
int		✓	✓	✗	✓
long		✓	✓	✗	✓
short		✓	✓	✗	✓
boolean[]		✗	✓	✗	✗
byte[]		✗	✓	✗	✗
char[]		✗	✓	✗	✗
double[]		✗	✓	✗	✗
float[]		✗	✓	✗	✗
int[]		✗	✓	✗	✗
long[]		✗	✓	✗	✗
short[]		✗	✓	✗	✗
java.lang.Boolean		✓	✓	✗	✓
java.lang.Byte		✓	✓	✗	✓
java.lang.Character		✓	✓	✗	✓
java.lang.Double		✓	✓	✗	✗
java.lang.Float		✓	✓	✗	✗
java.lang.Integer		✓	✓	✗	✓
java.lang.Long		✓	✓	✗	✓
java.lang.Short		✓	✓	✗	✓
java.lang.Boolean[]		✗	✓	✗	✗
java.lang.Byte[]		✗	✓	✗	✗

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.lang.Character[]		✘	✔	✘	✘
java.lang.Double[]		✘	✔	✘	✘
java.lang.Float[]		✘	✔	✘	✘
java.lang.Integer[]		✘	✔	✘	✘
java.lang.Long[]		✘	✔	✘	✘
java.lang.Short[]		✘	✔	✘	✘
java.lang.Number [4]		✔	✔	✘	✘
java.lang.Object		✘	✘	✘	✘
java.lang.String		✔	✔	✘	✔
java.lang.StringBuffer [3]	Extension	✘	✔	✘	✔
java.lang.String[]		✘	✔	✘	✘
java.math.BigDecimal		✔	✔	✘	✘
java.math.BigInteger		✔	✔	✘	✔
java.math.BigDecimal[]		✘	✔	✘	✘
java.math.BigInteger[]		✘	✔	✘	✘
java.sql.Date		✘	✘	✔	✔
java.sql.Time		✘	✘	✔	✔
java.sql.Timestamp		✘	✘	✔	✔
java.util.ArrayList		✘	✔	✔	✘
java.util.BitSet		✘	✘	✔	✘
java.util.Calendar	Extension	✘	✘	✔	✘

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.util.Collection		✗	✓	✓	✗
java.util.Currency		✗	✓	✗	✓
java.util.Date		✓	✓	✓	✓
java.util.Date[]		✗	✓	✗	✗
java.util.GregorianCalen [7]	Extension	✗	✗	✓	✗
java.util.HashMap		✗	✓	✓	✗
java.util.HashSet		✗	✓	✓	✗
java.util.Hashtable		✗	✓	✓	✗
java.util.LinkedHashMap [5]		✗	✓	✓	✗
java.util.LinkedHashSet [6]		✗	✓	✓	✗
java.util.LinkedList		✗	✓	✓	✗
java.util.List		✗	✓	✓	✗
java.util.Locale		✗	✓	✗	✓
java.util.Locale[]		✗	✓	✗	✗
java.util.Map		✗	✓	✓	✗
java.util.Properties	Extension	✗	✗	✓	✗
java.util.PriorityQueue	Extension	✗	✗	✓	✗
java.util.Queue	Extension	✗	✗	✓	✗
java.util.Set		✗	✓	✓	✗
java.util.SortedMap [2]	Extension	✗	✗	✓	✗
java.util.SortedSet [1]	Extension	✗	✗	✓	✗

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.util.Stack	Extension	✘	✘	✔	✘
java.util.TimeZone		✘	✘	✘	✔
java.util.TreeMap [2]		✘	✔	✔	✘
java.util.TreeSet [1]		✘	✔	✔	✘
java.util.UUID	Extension	✘	✘	✘	✔
java.util.Vector		✘	✔	✘	✘
java.awt.Color	Extension	✘	✘	✘	✘
java.awt.Point	Extension	✘	✘	✔	✘
java.awt.image.Buffered	Extension	✘	✘	✘	✘
java.net.URI	Extension	✘	✘	✘	✔
java.net.URL	Extension	✘	✘	✘	✔
java.io.Serializable		✘	✘	✘	✘
javax.jdo.spi.PersistenceCapable		✘	✘	✘	✔
javax.jdo.spi.PersistenceCapable[]		✘	✘	✘	✘
java.lang.Enum		✔	✔	✘	✔
java.lang.Enum[]		✘	✔	✘	✔

- [1] - java.util.SortedSet, java.util.TreeSet allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <collection>). The headSet, tailSet, subSet methods are only supported when using cached collections.
- [2] - java.util.SortedMap, java.util.TreeMap allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <map>). The headMap, tailMap, subMap methods are only supported when using cached containers.
- [3] - java.lang.StringBuffer dirty check mechanism is limited to immutable mode, it means, if you change a StringBuffer object field, you must reassign it to the owner object field to make sure changes are propagated to the database.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- [4] - java.lang.Number will be stored in a column capable of storing a BigDecimal, and will store to the precision of the object to be persisted. On reading back the object will be returned typically as a BigDecimal since there is no mechanism for determing the type of the object that was stored.
- [5] - java.util.LinkedHashMap treated as a Map currently. No List-ordering is supported.
- [6] - java.util.LinkedHashSet treated as a Set currently. No List-ordering is supported.
- [7] - java.util.Calendar can be stored into two columns (millisecs, Timezone) or into a single column (Timestamp). The single column option is not guaranteed to preserve the TimeZone of the input Calendar.

Note that support is available for persisting other types depending on the datastore to which you are persisting

- RDBMS GeoSpatial types via the DataNucleus RDBMS Spatial plugin
- JodaTime types for RDBMS datastores

If you have support for any additional types and would either like to contribute them, or have them listed here, let us know

DataNucleus allows you the luxury of being able to provide SCO support for your own Java types when using RDBMS datastores

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.3 Datastore Identity

Datastore Identity

With datastore identity you are leaving the assignment of id's to DataNucleus and your class will **not** have a field for this identity - it will be added to the datastore representation by DataNucleus. It is, to all extents and purposes a surrogate key that will have its own column in the datastore. To specify that a class is to use datastore identity with JDO, you add the following to the MetaData for the class.

```
<class name="MyClass" identity-type="datastore">
...
</class>
```

or using JDO2 annotations

```
@PersistenceCapable(identityType=IdentityType.DATASTORE)
public class MyClass
{
    ...
}
```

So you are specifying the identity-type as datastore. You don't need to add this because datastore is the default, so in the absence of any value, it will be assumed to be 'datastore'.

When you have an inheritance hierarchy, you should specify the identity type in the base class for the inheritance tree. This is then used for all persistent classes in the tree.

Generating identities

By choosing datastore identity you are handing the process of identity generation to the JDO implementation. This does not mean that you haven't got any control over how it does this. JDO 2 defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

Defining which one to use is a simple matter of adding a MetaData element to your classes definition, like this

```
<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="sequence" sequence="MY_SEQUENCE"/>
    ...
</class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="identity"/>
    ...
</class>
```

or using annotations, for example

```

@PersistenceCapable
@DatastoreIdentity(strategy="sequence", sequence="MY_SEQUENCE")
public class MyClass
{
    ...
}
```

Some of the datastore identity strategies require additional attributes, but the specification is straightforward.

See also :-

- Identity Generation Guide - strategies for generating ids
- MetaData reference for <datastore-identity> element
- Annotations reference for @DatastoreIdentity

Accessing the Identity

When using datastore identity, the class has no associated field so you can't just access a field of the class to see its identity - if you need a field to be able to access the identity then you should be usingapplication identity. There are, however, ways to get the identity for the datastore identity case, if you have the object.

```
Object id = pm.getObjectId(obj);
```

```
Object id = JDOHelper.getObjectId(obj);
```

You should be aware however that the "identity" is in a complicated form, and is not available as a simple integer value for example. Again, if you want an identity of that form then you should use application identity

DataNucleus Implementation

When implementing datastore identity all JDO implementations have to provide a public class that represents this identity. If you call pm.getObjectId(...) for a class using datastore identity you will be passed an object which, in the case of DataNucleus will be of type org.datanucleus.identity.OIDImpl. If you were to call "toString()" on this object you would get something like

```
1[OID]mydomain.myclass
This is made up of :-
    1 = identity number of this object
    class-name
```

The definition of this datastore identity is JDO implementation dependent. As a result you should not use the org.datanucleus.identity.OID class in your application if you want to remain implementation independent

DataNucleus allows you the luxury of being able to provide your own datastore identity class so you can have whatever formatting you want for identities.

3.4 Application Identity

Application Identity

With application identity you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key class (unless using SingleFieldIdentity, where one is provided for you), and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. With application identity the field(s) of the primary key will be present as field(s) of the class itself. To specify that a class is to use application identity, you add the following to the MetaData for the class.

```
<class name="MyClass" identity-type="application" objectid-class="MyIdClass">
    <field name="myPrimaryKeyField" primary-key="true"/>
    ...
</class>
```

For JDO2 we specify the identity-type and objectid-class. The objectid-class is the class defining the identity for this class. Alternatively, if we are using annotations

```
@PersistenceCapable(objectIdClass=MyIdClass.class)
public class MyClass
{
    @Persistent(primaryKey="true")
    private long myPrimaryKeyField;
}
```

When you have an inheritance hierarchy, you should specify the identity type and any primary-key fields in the base class for the inheritance tree. This is then used for all persistent classes in the tree.

See also :-

- MetaData reference for <field> element
- Annotations reference for @Persistent

Primary Key

Using application identity requires the use of a Primary Key class. In JDO 1.0 it was necessary to always provide this class. In JDO 2.0 an in-built class is available where the identity is defined in a single field. This is referred to as SingleFieldIdentity. DataNucleus supports this builtin class. Where the class has multiple fields that form the primary key a Primary Key class must be provided. In JPA1 when there is a single primary key field you dont need to specify the primary key class. If there are more than 1 id field then you define the id-class.

See also :-

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- Primary Key Guide - user-defined and built-in primary keys

Compound Identity

Where one of the fields that is primary-key of your class is a persistable object you have something known as **compound identity** since the identity of this class contains the identity of a related class. Please refer to the docs for Compound Identity

Generating identities

By choosing application identity you are controlling the process of identity generation for this class. This does not mean that you have a lot of work to do for this. JDO2 and JPA1 define many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

See also :-

- Identity Generation Guide - strategies for generating ids

Accessing the Identity

When using application identity, the class has associated field(s) that equate to the identity. As a result you can simply access the values for these field(s). Alternatively you could use a JDO identity-independent way

```
Object id = pm.getObjectId(obj);
```

```
Object id = JDOHelper.getObjectId(obj);
```

Changing Identities

JDO allows implementations to support the changing of the identity of a persisted object. This is an optional feature and DataNucleus doesn't currently support it.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.5 Nondurable Identity

Nondurable Identity

With nondurable identity your objects will not have a unique identity in the datastore. This type of identity is typically for log files, history files etc where you aren't going to access the object by key, but instead by a different parameter. In the datastore the table will typically not have a primary key. To specify that a class is to use nondurable identity with JDO2 you would add the following to the MetaData for the class.

```
<class name="MyClass" identity-type="nondurable">
...
</class>
```

or using annotations, for example

```
@PersistenceCapable(identityType=IdentityType.NONDURABLE)
public class MyClass
{
    ...
}
```

DataNucleus provides limited support for "nondurable" identity currently. Any class marked as "nondurable" will, for RDBMS datastores, have a table with no primary-key. It will support persistence of records into the datastore. Records can be deleted using SQL bulk delete statements. What is not currently supported is the ability to delete a particular object, or update a particular object without affecting all others with the same parameter values.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.6 Primary Keys

Primary Key Classes

As has been described in the application identity guide, when you choose application identity you are defining which fields of the class are part of the primary key, and you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key (PK) class, and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. You specify the primary key class like this

```
<class name="MyClass" identity-type="application" objectid-class="MyIdClass">
...
</class>
```

or using annotations

```
@PersistenceCapable(objectIdClass=MyIdClass.class)
public class MyClass
{
    ...
}
```

You now need to define the PK class to use. This is simplified for you because **if you have only one PK field then you dont need to define a PK class** and you only define it when you have a composite PK.

An important thing to note is that the PK can only be made up of fields of the following Java types

- Primitives : **boolean**, **byte**, **char**, **int**, **long**, **short**
- java.lang : **Boolean**, **Byte**, **Character**, **Integer**, **Long**, **Short**, **String**, **Enum**, StringBuffer
- java.math : **BigInteger**
- java.sql : **Date**, **Time**, **Timestamp**
- java.util : **Date**, **Currency**, **Locale**, TimeZone, UUID
- java.net : URI, URL
- javax.jdo.spi : **PersistenceCapable**

Note that the types in **bold** are JDO standard types. Any others are DataNucleus extensions and, as always, check the specific datastore docs to see what is supported for your datastore.

Single primary-key field

The simplest way of using application identity is where you have a single PK field, and in this case you use the JDO 2 SingleFieldIdentity Javadoc. mechanism. This provides a PrimaryKey for cases where you have a single PK field and you don't need to specify the objectid-class. Let's take an example

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class MyClass
{
    long id;
    String name;
    String description;
    ...
}

<class name="MyClass" identity-type="application">
    <field name="id" primary-key="true"/>
    <field name="name"/>
    <field name="description"/>
</class>
```

So we didnt specify the JDO "objectid-class". You will, of course, have to give the field a value before persisting the object, either by setting it yourself, or by using a value-strategy on that field.

If you need to create an identity of this form for use in querying via pm.getObjectById() then you can create the identities in the following way

```
For a "long" type :
javax.jdo.identity.LongIdentity id = new javax.jdo.identity.LongIdentity(myClass,
101);

For a "String" type :
javax.jdo.identity.StringIdentity id = new
javax.jdo.identity.StringIdentity(myClass, "ABCD");
```

We have shown an example above for type "long", but you can also use this for the following

```
short, Short        - javax.jdo.identity.ShortIdentity
int, Integer        - javax.jdo.identity.IntIdentity
long, Long          - javax.jdo.identity.LongIdentity
String              - javax.jdo.identity.StringIdentity
char, Character     - javax.jdo.identity.CharIdentity
byte, Byte          - javax.jdo.identity.ByteIdentity
java.util.Date      - javax.jdo.identity.ObjectIdentity
java.util.Currency - javax.jdo.identity.ObjectIdentity
java.util.Locale    - javax.jdo.identity.ObjectIdentity
```

Rules for User-Defined Primary Key classes

If you wish to use application identity and don't want to use the "SingleFieldIdentity" builtin PK classes then you must define a Primary Key class of your own. You can't use classes like java.lang.String, or java.lang.Long directly. You must follow these rules when defining your primary key class.

- the Primary Key class must be public
- the Primary Key class must implement Serializable

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- the Primary Key class must have a public no-arg constructor, which might be the default constructor
- the field types of all non-static fields in the Primary Key class must be serializable, and are recommended to be primitive, String, Date, or Number types
- all serializable non-static fields in the Primary Key class must be public
- the names of the non-static fields in the Primary Key class must include the names of the primary key fields in the JDO class, and the types of the common fields must be identical
- the equals() and hashCode() methods of the Primary Key class must use the value(s) of all the fields corresponding to the primary key fields in the JDO class
- if the Primary Key class is an inner class, it must be static
- the Primary Key class must override the toString() method defined in Object, and return a String that can be used as the parameter of a constructor
- the Primary Key class must provide a String constructor that returns an instance that compares equal to an instance that returned that String by the toString() method.
- the Primary Key class must be only used within a single inheritence tree.

Primary Key Example - Multiple Field

Here's an example of a composite (multiple field) primary key class

```
public class ComposedIdKey implements Serializable
{
    public String field1;
    public String field2;

    /**
     *  Default constructor.
     */
    public ComposedIdKey ()
    {
    }

    /**
     * String constructor.
     */
    public ComposedIdKey(String value)
    {
        StringTokenizer token = new StringTokenizer (value, "::");
        //className
        token.nextToken ();
        //field1
        this.field1 = token.nextToken ();
        //field2
        this.field2 = token.nextToken ();
    }

    /**
     * Implementation of equals method.
     */
    public boolean equals(Object obj)
    {
        if (obj == this)
        {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            return true;
        }
        if (!(obj instanceof ComposedIdKey))
        {
            return false;
        }
        ComposedIdKey c = (ComposedIdKey)obj;

        return field1.equals(c.field1) && field2.equals(c.field2);
    }

    /**
     *  Implementation of hashCode method that supports the
     *  equals-hashCode contract.
     */
    public int hashCode ()
    {
        return this.field1.hashCode() ^ this.field2.hashCode();
    }

    /**
     *  Implementation of toString that outputs this object id's PK values.
     */
    public String toString ()
    {
        return this.getClass().getName() + "::"  + this.field1 + "::" + this.field2;
    }
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.7 Fields/Properties

Persistent Fields or Properties

There are two distinct modes of persistence definition. The most common uses **fields**, whereas an alternative uses **properties**.

Persistent Fields

The most common form of persistence is where you have a **field** in a class and want to persist it to the datastore. With this mode of operation DataNucleus will persist the values stored in the fields into the datastore, and will set the values of the fields when extracting it from the datastore.

Requirement : you have a field in the class. This can be public, protected, private or package access, but cannot be static or final.

An example of how to define the persistence of a field is shown below

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    Date birthday;
}
```

So, using annotations, we have marked this class as persistent, and the field also as persistent. Using XML MetaData we would have done

```
<class name="MyClass">
    <field name="birthday" persistence-modifier="persistent"/>
</class>
```

Persistent Properties

A second mode of operation is where you have Java Bean-style getter/setter for a **property**. In this situation you want to persist the output from *getXXX* to the datastore, and use the *setXXX* to load up the value into the object when extracting it from the datastore.

Requirement : you have a property in the class with Java Bean getter/setter methods. These methods can be public, protected, private or package access, but cannot be static. The class must have BOTH getter AND setter methods.

An example of how to define the persistence of a property is shown below

```
@PersistenceCapable
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class MyClass
{
    @Persistent
    Date getBirthday()
    {
        ...
    }

    void setBirthday(Date date)
    {
        ...
    }
}
```

So, using annotations, we have marked this class as persistent, and the getter is marked as persistent. Using XML MetaData we would have done

```
<class name="MyClass">
    <property name="birthday" persistence-modifier="persistent"/>
</class>>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.8 Value Generation

Value generation

Fields of a class can either have the values set by you the user, or you can set DataNucleus to generate them for you. This is of particular importance with identity fields where you want unique identities. You can use this value generation process with any field in JDO, and with the identity field(s) in JPA. There are many different "strategies" for generating values, as defined by the JDO specifications, and also some DataNucleus extensions. Some strategies are specific to a particular datastore, and some are generic. You should choose the strategy that best suits your target datastore. The available strategies for JDO are :-

- native - this is the default and allows DataNucleus to choose the most suitable for the datastore
- sequence - this uses a datastore sequence (if supported by the datastore)
- identity - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- increment - this is datastore neutral and increments a sequence value using a table.
- uuid-string - this is a UUID in string form
- uuid-hex - this is a UUID in hexadecimal form
- datastore-uuid-hex - UUID in hexadecimal form using datastore capabilities
- max - uses a max(column)+1 method
- auid - provides a pure UUID following the OpenGroup standard
- timestamp - creates a java.sql.Timestamp of the current time
- timestamp-value - creates a long (millisecs) of the current time
- user-supplied value generators - allows you to hook in your own identity generator

See also :-

- JDO MetaData reference for <class>
- JDO MetaData reference for <datastore-identity>
- JDO MetaData reference for <field>
- JDO Annotation reference for @DatastoreIdentity
- JDO Annotation reference for @Persistent

native

With this strategy DataNucleus will choose the most appropriate strategy for the datastore being used. If you also specify the 'sequence' name attribute and the datastore supports sequences then "sequence" strategy would be used. Otherwise it will always choose "increment" strategy.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

sequence

A sequence is a user-defined database function that generates a sequence of unique numeric ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. DataNucleus supports sequences for the following datastores:

- Oracle
- PostgreSQL
- SAP DB
- DB2
- Firebird
- HSQLDB
- H2
- DB4O

To configure a class to use either of these generation methods with datastore identity you simply add this to the class' Meta-Data

```
<class name="myclass" ... >
    <datastore-identity strategy="sequence" sequence="yourseq"/>
    ...
    <sequence name="yourseq" datastore-sequence="YOUR_SEQUENCE_NAME"/>
</class>
```

or using annotations

```
@PersistenceCapable
@DatastoreIdentity(strategy="sequence", sequence="yourseq"/>
@Sequence(name="yourseq", datastore-sequence="YOUR_SEQUENCE_NAME"/>
public class MyClass
```

You replace "YOUR_SEQUENCE_NAME" with your sequence name. To configure a class to use either of these generation methods using application identity you would add the following to the class' Meta-Data

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="sequence"
sequence="yourseq"/>
    ...
    <sequence name="yourseq" datastore-sequence="YOUR_SEQUENCE_NAME"/>
</class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

or using annotations

```
@PersistenceCapable
@Sequence(name="yourseq", datastore-sequence="YOUR_SEQUENCE_NAME"/>
public class MyClass
{
    @Persistent(valueStrategy="sequence", sequence="yourseq"/>
    private long myfield;
    ...
}
```

If the sequence does not yet exist in the database at the time DataNucleus needs a new unique identifier, a new sequence is created in the database based on the JDO Meta-Data configuration. Additional properties for configuring sequences are set in the JDO Meta-Data, see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

Property	Description	Required
key-cache-size	number of unique identifiers to cache in the PersistenceManagerFactory instance. Notes: 1. This setting SHOULD match the *key-start-with* setting value if *key-start-with* is provided, otherwise it can cause duplicate keys errors when inserting new objects into the database. 2. The keys are pre-allocated, cached and used on demand. If *key-cache-size* is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.	No. Defaults to 1.
key-min-value	determines the minimum value a sequence can generate	No
key-max-value	determines the maximum value a sequence can generate	No
key-start-with	the initial value for the sequence	No
key-increment-by	specifies which value is added to the current sequence value to create a new value. default is 1	No
key-database-cache-size	specifies how many sequence numbers are to be preallocated and stored in memory for faster access. This is an optimization feature provided by the database	No
sequence-catalog-name	Name of the catalog where the sequence is.	No.
sequence-schema-name	Name of the schema where the sequence is.	No.

This value generator will generate values unique across different JVMs

identity

Auto-increment/identity/serial are primary key columns that are populated when a row is inserted in the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

table. These use the databases own keywords on table creation and so rely on having the table structure either created by DataNucleus or having the column with the necessary keyword.

DataNucleus supports auto-increment/identity/serial keys for many databases including :

- DB2 (IDENTITY)
- MySQL (AUTOINCREMENT)
- MSSQL (IDENTITY)
- Sybase (IDENTITY)
- HSQLDB (IDENTITY)
- H2 (IDENTITY)
- PostgreSQL (SERIAL)

This generation strategy should only be used if there is a single "root" table for the inheritance tree. If you have more than 1 root table (e.g using subclass-table inheritance) then you should choose a different generation strategy

For a class using datastore identity you need to set the *strategy* attribute. You can configure the Meta-Data for the class something like this (replacing 'myclass' with your class name) :

```
<class name="myclass">
    <datastore-identity strategy="identity"/>
    ...
</class>
```

For a class using application identity you need to set the *value-strategy* attribute on the primary key field. You can configure the Meta-Data for the class something like this (replacing 'myclass' and 'myfield' with your class and field names) :

```
<class name="myclass" identity-type="application"
objectid-class="myprimarykeyclass">
    <field name="myfield" primary-key="true" value-strategy="identity"/<
    ...
</class>
```

Please be aware that if you have an inheritance tree with the base class defined as using "identity" then the column definition for the PK of the base table will be defined as "AUTO_INCREMENT" or "IDENTITY" or "SERIAL" (dependent on the RDBMS) and all subtables will NOT have this identifier added to their PK column definitions. This is because the identities are assigned in the base table (since all objects will have an entry in the base table).

This value generator will generate values unique across different JVMs

increment

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This method is database neutral and uses a sequence table that holds an incrementing sequence value. The unique identifier value returned from the database is translated to a java type: java.lang.Long. This strategy will work with any datastore. This method require a sequence table in the database and creates one if doesn't exist.

To configure a datastore identity class to use this generation method you simply add this to the classes Meta-Data.

```
<class name="myclass" ... >
    <datastore-identity strategy="increment"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' Meta-Data. If your class is in an inheritance tree you should define this for the base class only.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="increment"/>
    ...
</class>>
```

Additional properties for configuring this generator are set in the JDO Meta-Data, see the available properties below. Unsupported properties are silently ignored by DataNucleus.

Property	Description	Required
key-initial-value	First value to be allocated.	No. Defaults to 1
key-cache-size	number of unique identifiers to cache. The keys are pre-allocated, cached and used on demand. If *key-cache-size* is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.	No. Defaults to 5
sequence-table-basis	Whether to define uniqueness on the base class name or the base table name. Since there is no "base table name" when the root class has "subclass-table" this should be set to "class" when the root class has "subclass-table" inheritance	No. Defaults to *class*, but the other option is table
sequence-name	name for the sequence (overriding the "sequence-table-basis" above). The row in the table will use this in the PK column	No
sequence-table-name	Table name for storing the sequence.	No. Defaults to *SEQUENCE_TABLE*
sequence-catalog-name	Name of the catalog where the table is.	No.
sequence-schema-name	Name of the schema where the table is.	No.
sequence-name-column-name	Name for the column that represent sequence names.	No. Defaults to *SEQUENCE_NAME*

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Property	Description	Required
sequence-nextval-column-name	Name for the column that represent incremeting sequence values.	No. Defaults to *NEXT_VAL*
table-name	Name of the table whose column we are generating the value for (used when we have no previous sequence value and want a start point.	No.
column-name	Name of the column we are generating the value for (used when we have no previous sequence value and want a start point.	No.

This value generator will generate values unique across different JVMs (from DataNucleus 1.1.3)

uuid-string (JDO2)

This generator creates identities with 16 characters in string format. The identity contains the IP address of the local machine where DataNucleus is running, as well as other necessary components to provide uniqueness across time.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would break performance limits. It doesn't require datastore access to generate the identities and so has performance benefits over some of the other generators.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="uuid-string"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="uuid-string"/>
    ...
</class>
```

uuid-hex (JDO2)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This generator creates identities with 32 characters in hexadecimal format. The identity contains the IP address of the local machine where DataNucleus is running, as well as other necessary components to provide uniqueness across time.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would break performance limits. It doesn't require datastore access to generate the identities and so has performance benefits over some of the other generators.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="uuid-hex"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="uuid-hex"/>
    ...
</class>
```

datastore-uuid-hex

This method is like the "uuid-hex" option above except that it utilises datastore capabilities to generate the UUIDHEX code. Consequently this only works on some RDBMS (MSSQL, MySQL). The disadvantage of this strategy is that it makes a call to the datastore for each new UUID required. The generated UUID is in the same form as the AUID strategy where identities are generated in memory and so the AUID strategy is the recommended choice relative to this option.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="datastore-uuid-hex"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="datastore-uuid-hex"/>
    ...
</class>
```

max

This method is database neutral and uses the "select max(column) from table" + 1 strategy to create unique ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. It is however not recommended by DataNucleus since it makes a DB call for every record to be inserted and hence is inefficient. Each DB call will run a scan in all table contents causing contention and locks in the table. We recommend the use of either Sequence or Identity based value generators (see below) - which you use would depend on your RDBMS.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="max"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="max"/>
    ...
</class>
```

This value generator will **NOT** guarantee to generate values unique across different JVMs. This is because it will select the "max+1" and before creating the record another thread may come in and insert one.

auid

This generator uses a Java implementation of DCE UUIDs to create unique identifiers without the overhead of additional database transactions or even an open database connection. The identifiers are Strings of the form "LLLLLLLL-MMMM-HHHH-CCCC-NNNNNNNNNNNN" where 'L', 'M', 'H', 'C' and 'N' are the DCE UUID fields named time low, time mid, time high, clock sequence and node.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

synchronizing the concurrent creation of unique identifiers through the database would break performance limits.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="auid"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="auid"/>
    ...
</class>
```

This value generator will generate values unique across different JVMs

timestamp

This method will create a java.sql.Timestamp of the current time (at insertion in the datastore).

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="timestamp"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="timestamp"/>
    ...
</class>
```

timestamp-value

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This method will create a long of the current time in millisecs (at insertion in the datastore).

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="timestamp-value"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="timestamp-value"/>
    ...
</class>
```

Standalone ID generation

This section describes how to use the DataNucleus Value Generator API for generating unique keys for objects outside the DataNucleus (JDO) runtime. DataNucleus defines a framework for identity generation and provides many builtin strategies for identities. You can make use of the same strategies described above but for generating identities manually for your own use. The process is described below

The DataNucleus Value Generator API revolves around 2 classes. The entry point for retrieving generators is the ValueGenerationManager. This manages the appropriate ValueGenerator classes. Value generators maintain a block of cached ids in memory which avoid reading the database each time it needs a new unique id. Caching a block of unique ids provides you the best performance but can cause "holes" in the sequence of ids for the stored objects in the database.

Let's take an example. Here we want to obtain an identity using the TableGenerator ("increment" above). This stores identities in a datastore table. We want to generate an identity using this. Here is what we add to our code

```
PersistenceManagerImpl pm = (PersistenceManagerImpl) ... // cast your pm to impl ;

// Obtain a ValueGenerationManager
ValueGenerationManager mgr = new ValueGenerationManager();

// Obtain a ValueGenerator of the required type
Properties properties = new Properties();
properties.setProperty("table-name", "GLOBAL"); // Use a global sequence number (for
all tables)
ValueGenerator generator = mgr.createValueGenerator("MyGenerator",
    org.datanucleus.store.rdbms.valuegenerator.TableGenerator.class, props,
pm.getStoreManager(),
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
                new ValueGenerationConnectionProvider()
                {
                    RDBMSManager rdbmsManager = null;
                    ManagedConnection con;
                    public ManagedConnection retrieveConnection()
                    {
                        rdbmsManager = (RDBMSManager) pm.getStoreManager();
                        try
                        {
                            // important to use TRANSACTION_NONE like DataNucleus
does
                            con =
rdbmsManager.getConnection(Connection.TRANSACTION_NONE);;
                            return con;
                        }
                        catch (SQLException e)
                        {
                            logger.error("Failed to obtain new DB connection for
identity generation!");
                            throw new RuntimeException(e);
                        }
                    }
                    public void releaseConnection()
                    {
                        try
                        {
                            con.close();
                            con = null;
                        }
                        catch (DataNucleusException e)
                        {
                            logger.error("Failed to close DB connection for identity
generation!");
                            throw new RuntimeException(e);
                        }
                        finally
                        {
                            rdbmsManager = null;
                        }
                    }
                });

// Retrieve the next identity using this strategy
Long identifier = (Long)generator.next();
```

Some ValueGenerators are specific to RDBMS datastores, and some are generic, so bear this in mind when selecting and adding your own.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.9 Sequences

JDO Datastore Sequences

Particularly when specifying the identity of an object, sequences are a very useful facility. DataNucleus supports the automatic assignment of sequence values for object identities. However such sequences may also have use when a user wishes to assign such identity values themselves, or for other roles within an application. JDO 2 defines an interface for sequences for use in an application - known as Sequence. . There are 2 forms of "sequence" available through this interface - the ones that DataNucleus provides utilising datastore capabilities, and ones that a user provides using something known as a "factory class".

DataNucleus Sequences

DataNucleus internally provides 2 forms of sequences. When the underlying datastore supports native sequences, then these can be leveraged through this interface. Alternatively, where the underlying datastore doesn't support native sequences, then a table-based incrementing sequence can be used. The first thing to do is to specify the Sequence in the Meta-Data for the package requiring the sequence. This is done as follows

```
<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>

        <sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ"
strategy="contiguous"/>
        <sequence name="ProductSequenceNontrans"
datastore-sequence="PRODUCT_SEQ_NONTRANS" strategy="nontransactional"/>
    </package>
</jdo>
```

So we have defined two Sequences for the package MyPackage. Each sequence has a symbolic name that is referred to within JDO (within DataNucleus), and it has a name in the datastore. The final attribute represents whether the sequence is transactional or not.

All we need to do now is to access the Sequence in our persistence code in our application. This is done as follows

```
PersistenceManager pm = pmf.getPersistenceManager();

Sequence seq = pm.getSequence("MyPackage.ProductSequence");
```

and this Sequence can then be used to provide values.

ALL RIGHTS RESERVED

```
long value = seq.nextValue();
```

Please be aware that when you have a Sequence declared with a strategy of "contiguous" this means "transactional contiguous" and that you need to have a Transaction open when you access it.

A DataNucleus extension to this capability allows the user some control over the underlying datastore sequence being used. This is specified using <extension>. The underlying sequences used by DataNucleus here are the "SequenceTableValueGenerator" and "SequenceTableGenerator" as described in the Identity Generation guide. So we can do

```
        <sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ"
strategy="contiguous">
            <extension vendor-name="datanucleus" key="key-cache-size" value="10"/>
        </sequence>
```

which will allocate 10 new sequence values each time the allocated sequence values is exhausted.

Factory Class Sequences

It is equally possible to provide your own Sequence capability using a factory class. This is a class that creates an implementation of the JDO Sequence. Let's give an example of what you need to provide. Firstly you need an implementation of the JDO Sequence interface, so we define ours like this

```
public class SimpleSequence implements Sequence
{
    String name;
    long current = 0;

    public SimpleSequence(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public Object next()
    {
        current++;
        return new Long(current);
    }

    public long nextValue()
    {
        current++;
        return current;
    }

    public void allocate(int arg0)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    {
    }

    public Object current()
    {
        return new Long(current);
    }

    public long currentValue()
    {
        return current;
    }
}
```

So our sequence simply increments by 1 each call to next(). The next thing we need to do is provide a factory class that creates this Sequence. This factory needs to have a static newInstance method that returns the Sequence object. We define our factory like this

```
package org.datanucleus.samples.sequence;

import javax.jdo.datastore.Sequence;

public class SimpleSequenceFactory
{
    public static Sequence newInstance()
    {
        return new SimpleSequence("MySequence");
    }
}
```

and now we define our MetaData like this

```
<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>

        <sequence name="ProductSequenceFactory" strategy="nontransactional"
            factory-class="org.datanucleus.samples.sequence.SimpleSequenceFactory"/>
    </package>
</jdo>
```

So now we can call

```
PersistenceManager pm = pmf.getPersistenceManager();

Sequence seq = pm.getSequence("MyPackage.ProductSequenceFactory");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

3.10 JDO MetaData

JDO Metadata Overview

JDO requires the persistence of classes to be defined via Metadata. This Metadata can be provided in the following forms

- XML : the traditional mechanism, with XML files containing information for each class to be persisted. As a further complication you can define basic persistence metadata for a class in one file, and then ORM metadata for that class in a separate file (since the ORM metadata is specific to a certain datastore).
- Annotations : using JDK1.5+ annotations in the classes to be persisted
- API : a programmatic API allowing definition of which classes are to be persisted at runtime

Metadata priority

JDO defines the priority order for metadata as being

- API Metadata
- ORM XML Metadata
- JDO XML Metadata
- Annotations

So if a class has Metadata defined by API then that will override all other Metadata. If a class has annotations and JDO XML Metadata then the XML Metadata will take precedence over the annotations (or rather be merged on top of the annotations).

XML Metadata loading

JDO expects the XML Meta-Data to be specified in a file or files in particular locations in the file system. For example, if you have a class *com.mycompany.sample.myexample*, JDO will look for any of the following files until it finds one (in the order stated) :-

```
META-INF/package.jdo
WEB-INF/package.jdo
package.jdo
com/package.jdo
com/mycompany/package.jdo
com/mycompany/sample/package.jdo
com/mycompany/sample/myexample.jdo
```

In addition, for this example, DataNucleus allows the previous JDO 1.0.0 alternatives of

```
com.jdo
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
com/mycompany.jdo
com/mycompany/sample.jdo
```

In addition to the above, you can split your MetaData definitions between JDO MetaData files. For example if you have the following classes

```
com/mycompany/A.java
com/mycompany/B.java
com/mycompany/C.java
com/mycompany/app1/D.java
com/mycompany/app1/E.java
```

You could define the MetaData for these 5 classes in many ways -- for example put all class definitions in com/mycompany/package.jdo, or put the definitions for D and E in com/mycompany/app1/package.jdo and the definitions for A, B, C in com/mycompany/package.jdo, or have some in their class named MetaData files e.g com/mycompany/app1/A.jdo, or a mixture of the above. DataNucleus will always search for the MetaData file containing the class definition for the class that it requires.

XML Metadata validation

By default any XML Metadata will be validated for accuracy when loading it. Obviously XML is defined by a DTD or XSD schema and so should follow that. You can turn off such validations by setting the persistence property **datanucleus.metadata.validate** to false when creating your PMF. Note that this only turns off the XML strictness validation, and *not* the checks on inconsistency of specification of relations etc.

XML ORM Metadata usage

You can use ORM metadata to override particular datastore-specific things like table and column names. If your application doesn't make use of ORM metadata then you could turn off the searches for ORM Metadata files when a class is loaded up. You do this with the persistence property **datanucleus.metadata.supportORM** setting it to false.

Metadata discovery at class initialisation

JDO provides a mechanism whereby when a class is initialised (by the ClassLoader) any PersistenceManagerFactory is notified of its existence, and its Metadata can be loaded. This is enabled by the enhancement process. If you decided that you maybe only wanted some classes present in one PMF and other classes present in a different PMF then you can disable this and leave it to DataNucleus to discover the Metadata when operations are performed on that PMF. The persistence property to define to disable this is **datanucleus.metadata.autoregistration** (setting it to false).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.10.1 JDO XML

JDO XML Meta-Data Reference

One of the types of Metadata that JDO accepts is in XML form. As described in the Metadata Overview this has to be contained in files with particular filenames in particular locations (relative to the name of the class), and that this metadata is *discovered* at runtime. This page defines the format of the XML Metadata.

JDO MetaData has the following format (obviously only some of the elements/attributes are applicable to an ORM XML file). Please refer to the JDO XSD for precise details. What follows provides a reference guide to MetaData elements.

- jdo
    - package
        - class
            - implements
            - datastore-identity
                - column
                - extension
            - primary-key
                - column
            - inheritance
                - discriminator
                    - column
                - join
                    - column
            - version
                - column
                - extension
            - join
                - column
            - foreign-key
                - column
                - field
                - property

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- index
  - column
  - field
  - property
- unique
  - column
  - field
  - property
- field
  - collection
    - extension
  - map
    - extension
  - array
  - join
    - primary-key
    - column
  - embedded
    - field
      - column
  - element
    - column
  - key
    - column
  - value
    - column
  - order
    - column
    - extension
  - column
    - extension
  - foreign-key

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

        - column
    - index
        - column
    - unique
        - column
    - extension
- property
    - collection
        - extension
    - map
        - extension
    - array
    - join
        - primary-key
        - column
    - embedded
        - field
            - column
    - element
        - column
    - key
        - column
    - value
        - column
    - order
        - column
    - column
        - extension
    - foreign-key
        - column
    - index
        - column

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

        - unique
            - column
        - extension
    - fetch-group
        - field
    - query
- sequence
    - extension
- fetch-plan
- extension

- extension

Metadata for package tag

These are attributes within the <package> tag (jdo/package). This is used to denote a package, and all of the <class> elements that follow are in this Java package.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the java package	
catalog	Name of the catalog in which to persist the classes in this package. See also the property name "javax.jdo.mapping.Catalog" in the PMF Guide.	
schema	Name of the schema in which to persist the classes in this package. See also the property name "javax.jdo.mapping.Schema" in the PMF Guide.	

Metadata for class tag

These are attributes within the <class> tag (jdo/package/class). This is used to define the persistence definition for this class.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the class to persist	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
identity-type	The identity type, specifying whether they are uniquely provided by the JDO implementation (datastore identity), accessible fields in the object (application identity), or not at all (nondurable identity). DataNucleus only supports nondurable identity for SQL views.	datastore, application, nondurable
objectid-class	The class name of the primary key. When using application identity.	
requires-extent	Whether the JDO implementation must provide an Extent for this class.	true, false
detachable	Whether the class is detachable from the persistence graph.	true, false
embedded-only	Whether this class should only be stored embedded in the tables for other classes.	true, false
persistence-modifier	What type of persistability type this class exhibits. Please refer to JDO Class Types.	persistence-capable | persistence-aware | non-persistent
persistence-capable-superclass	Class name of superclass that is persistent capable. This is deprecated in JDO2 and you no longer need to specify it, leaving it to DataNucleus to determine if there is a superclass that is PersistenceCapable.	
catalog	Name of the catalog in which to persist the class. See also the property name "javax.jdo.mapping.Catalog" in the PMF Guide.	
schema	Name of the schema in which to persist the class. See also the property name "javax.jdo.mapping.Schema" in the PMF Guide.	
table	Name of the table/view in which to persist the class. See also the property name "datanucleus.identifier.case" in the Persistence Properties Guide.	
cacheable	Whether the class can be cached in a Level 2 cache. **From JDO2.2**	true | false

Metadata for implements tag

These are attributes within the <implements> tag (jdo/package/class/implements). This is used when the <class> implements interfaces that are used as field types in other classes. **This is deprecated in JDO2.1 since it can be determined from reflection on the class**

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the interface being implemented (fully qualified, else will look in the same package)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for datastore-identity tag

These are attributes within the <datastore-identity> tag (jdo/package/class/datastore-identity). This is used when the <class> to which this pertains uses datastore identity. It is used to define the precise definition of datastore identity to be used. This element can contain column sub-elements allowing definition of the column details where required - these are optional.

Attribute	Description	Values
Standard (JDO) Tags		
strategy	Strategy for datastore identity generation for this class. native allows DataNucleus to choose the most suitable for the datastore. sequence will use a sequence (specified by the attribute sequence) - if supported by the datastore. increment will use the id values in the datastore to decide the next id. uuid-string will use a UUID string generator (16-characters). uuid-hex will use a UUID string generator (32-characters). identity will use a datastore inbuilt auto-incrementing types. auid is a DataNucleus extension, that is an almost universal id generator (best possible derivate of a DCE UUID). max is a DataNucleus extension, that uses "select max(column)+1 from table" for the identity. timestamp is a DataNucleus extension, providing the current timestamp. timestamp-value is a DataNucleus extension, providing the current timestamp millisecs. [other values] to utilise user-supplied DataNucleus value generator plugins.	native | sequence | increment | identity | uuid-string | uuid-hex | auid | max | timestamp | timestamp-value | [other values]
sequence	Name of the sequence to use to generate identity values, when using a strategy of sequence. Please see also the class extension tags for controlling the sequence.	
column	Name of the column used for the datastore identity for this class.	

These are attributes within the <extension> tag (jdo/package/class/datastore-identity/extension). These are for controlling the generation of ids when in datastore identity mode.

Attribute	Description	Values
Extension (JDO) Tags		
sequence-table-basis	This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the "increment" strategy, and sometimes by "native"). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment	**class** | table

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
sequence-catalog-name	The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.	
sequence-schema-name	The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.	
sequence-table-name	The table used to store sequences for use by value generators. See Value Generation.	**SEQUENCE_TABLE**
sequence-name-column-name	The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.	**SEQUENCE_NAME**
sequence-nextval-column-name	The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.	**NEXT_VAL**
key-min-value	The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.	
key-max-value	The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.	
key-start-with	The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.	
key-increment-by	The increment value for use by value generators. Keys will be incremented by this value when being generated. See Value Generation.	
key-database-cache-size	The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	
key-cache-size	The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	

Metadata for primary-key tag

These are attributes within the <primary-key> tag (jdo/package/class/primary-key or class/field/join/primary-key). It is used to specify the name of the primary key constraint in the datastore during the schema generation process. When used under <join> it specifies that the join table has a primary-key.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the primary key constraint.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
column	Name of the column to use for the primary key	

Metadata for inheritance tag

These are attributes within the <inheritance> tag (jdo/package/class/inheritance). It is used when this class is part of an inheritance tree, and to denote how the class is stored in the datastore since there are several ways (strategies) in which it can be stored.

Attribute	Description	Values
Standard (JDO) Tags		
strategy	Strategy for inheritance of this class. Please refer to the Inheritance Guide. Note that "complete-table" is a DataNucleus extension to JDO2	new-table, subclass-table, superclass-table, complete-table

Metadata for discriminator tag

These are attributes within the <discriminator> tag (jdo/package/class/inheritance/discriminator). This is used to define a discriminator column that is used when this class is stored in the same table as another class in the same inheritance tree. The discriminator column will contain a value for objects of this class, and different values for objects of other classes in the inheritance tree.

Attribute	Description	Values
Standard (JDO) Tags		
strategy	Strategy for the discrimination column	value-map | class-name | none
value	Value for the discrimination column	
column	Name for the discrimination column	
indexed	Whether the discriminator column should be indexed. This is to be specified when defining index information	true | false | unique

Metadata for version tag

These are attributes within the <version> tag (jdo/package/class/version). This is used to define whether and how this class is handled with respect to optimistic transactions.

Attribute	Description	Values
Standard (JDO) Tags		

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
strategy	Strategy for versioning of this class. The "version-number" mode uses an incremental numbered value, and the "date-time" mode uses a java.sql.Timestamp value. state-image isn't currently supported.	state-image, date-time, version-number
column	Name of the column in the datastore to store this field	
indexed	Whether the version column should be indexed. This is to be specified when defining index information	true | false | unique

These are attributes within the <extension> tag (jdo/package/class/version/extension).

Attribute	Description	Values
Extension (JDO) Tags		
field-name	This extension allows you to define a field that will be used to contain the version of the object. It is populated by DataNucleus at persist. See JDO Versioning	

Metadata for query tag

These are attributes within the <query> tag (jdo/package/class/query). This element is used to define any "named queries" that are to be available for this class. This element contains the query single-string form as its content.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the query. This name is mandatory and is used in calls to pm.newNamedQuery(). Has to be unique for this class.	
language	Query language to use. Some datastores offer other languages	JDOQL | SQL | JPQL
unique	Whether the query is to return a unique result (only for SQL queries).	true | **false**
result-class	Class name of any result class (only for SQL queries).	

Metadata for field tag

These are attributes within the <field> tag (jdo/package/class/field). This is used to define the persistence behaviour of the fields of the class to which it pertains. Certain types of fields are, by default, persisted. This element can be used to change the default behaviour and maybe not persist a field, or to persist something that normally isn't persisted. It is used, in addition, to define more details about how

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

the field is persisted in the datastore.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the field.	
persistence-modifier	The persistence-modifier specifies how JDO manage each field in your persistent class. There are three options: persistent, transactional and none. • persistent means that your field will managed by JDO and stored in the database on transaction commit. • transactional means that your field will managed by JDO but not stored in the database. Transactional fields values will be saved by JDO when you start your transaction and restored when you roll back your transaction. • none means that your field will not be managed by JDO.	persistent, transactional, none
primary-key	Whether the field is part of any primary key (if using application identity).	true, false
null-value	How to treat null values of persistent fields during storage. Valid options are "exception", "default", "none" (where "none" is the default).	exception, default, none
default-fetch-group	Whether this field is part of the default fetch group for the class. Defaults to true for non-key fields of primitive types, java.util.Date, java.lang.*, java.math.*, etc.	true, false
embedded	Whether this field should be stored, if possible, as part of the object instead as its own object in the datastore. This defaults to true for primitive types, java.util.Date, java.lang.*, java.math.* etc and false for PersistenceCapable, reference (Object, Interface) and container types.	true, false
serialized	Whether this field should be stored serialised into a single column of the table of the containing object.	true, false
dependent	Whether the field should be used to check for dependent objects, and to delete them when this object is deleted. In other words cascade delete capable.	true, false
mapped-by	The name of the field at the other end of a relationship. Used by 1-1, 1-N, M-N to mark a relation as bidirectional.	
value-strategy	The strategy for populating values to this field. Is typically used for generating primary key values. See the definitions under "datastore-identity".	native | sequence | increment | identity | uuid-string | uuid-hex | auid | max | timestamp | timestamp-value | [other values]
sequence	Name of the sequence to use to generate values, when using a strategy of sequence. Please see also the class extension tags for controlling the sequence.	
recursion-depth	The depth that will be recursed when this field is self-referencing. Should be used alongside FetchPlan.setMaxFetchDepth() to control the objects fetched.	-1, 1, 2, ... (integer)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
field-type	Used to specify a more restrictive type than the field definition in the class. This might be required in order to map the field to the datastore. To be portable, specify the name of a single type that is itself able to be mapped to the datastore (e.g. a field of type Object can specify field-type="Integer").	
indexed	Whether the column(s) for this field should be indexed. This is to be specified when defining index information	true | false | unique
table	Table name to use for any join table overriding the default name provided by DataNucleus. This is used either for 1-N relationships with a join table or for Secondary Tables. See also the property name "datanucleus.identifier.case" in the Persistence Properties Guide.	
column	Column name to use for this field (alternative to specifying column sub-elements if only one column).	
delete-action	The foreign-key delete action. This is a shortcut to specifying foreign key information. Please refer to the <foreign-key> element for full details.	cascade | restrict | null | default | none
cacheable	Whether the field/property can be cached in a Level 2 cache. **From JDO2.2**	true | false

These are attributes within the <extension> tag (jdo/package/class/field/extension).

Attribute	Description	Values
Extension (JDO) Tags		
cascade-persist	JDO defines that when an object is persisted then all fields will also be persisted using "persistence-by-reachability". This extension allows you to turn off the persistence of a field relation.	**true** | false
cascade-update	JDO defines that when an object is updated then all fields containing PersistenceCapable objects will also be updated using "persistence-by-reachability". This extension allows you to turn off the update of a field relation.	**true** | false
cascade-refresh	When calling PersistenceManager.refresh() only fetch plan fields of the passed object will be refreshed. Setting this to true will refresh the fields of related PC objects in this field	true | **false**
allows-null	When the field is a collection by default it will not be allowed to have nulls present but you can allow them by setting this DataNucleus extension tag	true | **false**
insertable	Whether this field should be supplied when inserting into the datastore.	**true** | false
updateable	Whether this field should be supplied when updating the datastore.	**true** | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
adapter-column-name	In some situations DataNucleus will add a special datastore column to a join table so that collections can allow the storage of duplicate elements. This extension allows the specification of the column name to be used. This should be specified within the field at the collection end of the relationship. JDO2 doesnt allow a standard place for such a specification and so is an extension tag.	**INTEGER_IDX**
implementation-classes	Used to define the possible classes implementing this interface/Object field. This is used to limit the possible tables that this is a foreign key to (when this field is specified as an interface/Object in the class). Value should be comma-separated list of fully-qualified class names	
key-implementation-classes	Used to define the possible classes implementing this interface/Object key. This is used to limit the possible tables that this is a foreign key to (when this key is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names	
value-implementation-classes	Used to define the possible classes implementing this interface/Object value. This is used to limit the possible tables that this is a foreign key to (when this value is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names	
strategy-when-notnull	This is to be used in conjunction with the "value-strategy" attribute. Default JDO2 behaviour when you have a "value-strategy" defined for a field is to always create a strategy value for that field regardless of whether you have set the value of the field yourself. This extension allows you to only apply the strategy if the field is null at persistence. This extension has no effect on primitive field types (which can't be null) and the value-strategy will always be applied to such fields.	**true** | false
relation-discriminator-column	Name of a column to use for discrimination of the relation used by objects stored. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for	**RELATION_DISCRIM**
relation-discriminator-pk	Whether the column added for the discrimination of relations is to be part of the PK when using a join table.	true | **false**
relation-discriminator-value	Value to use in the relation discriminator column for objects of this fields relation. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.	Fully-qualified class name

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
select-function	Permits to use a function when fetching contents from the database. A ? (question mark) is mandatory to have and will be replaced by the column name when generating the SQL statement. For example to specify a value of *UPPER(?)* will convert the field value to upper case on a datastore that supports that UPPER function.	
insert-function	Permits to use a function when inserting into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of *TRIM(?)* will trim the field value on a datastore that supports that TRIM function.	
update-function	Permits to use a function when updating into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of *FUNC(?)* will perform "FUNC" on the field value on a datastore that supports that FUNC function.	
sequence-table-basis	This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the "increment" strategy, and sometimes by "native"). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment	**class** | table
sequence-catalog-name	The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.	
sequence-schema-name	The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.	
sequence-table-name	The table used to store sequences for use by value generators. See Value Generation.	**SEQUENCE_TABLE**
sequence-name-column-name	The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.	**SEQUENCE_NAME**
sequence-nextval-column-name	The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.	**NEXT_VAL**
key-min-value	The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.	
key-max-value	The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.	
key-start-with	The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
key-increment-by	The increment value for use by value generators. Keys will be incremented by this value when being generated. See Value Generation.	
key-database-cache-size	The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	
key-cache-size	The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	
mapping-class	Specifies the mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus mapping class and provide their own mapping class for this field.	Fully-qualified class name

Metadata for property tag

These are attributes within the <property> tag (jdo/package/class/property). This is used to define the persistence behaviour of the Java Bean properties of the class to which it pertains. This element can be used to change the default behaviour and maybe not persist a property, or to persist something that normally isn't persisted. It is used, in addition, to define more details about how the property is persisted in the datastore.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the property. The "name" of a property is obtained by taking the getXXX, setXXX method names and using the XXX and making the first letter lowercase.	
persistence-modifier	The persistence-modifier specifies how to manage each property in your persistent class. There are three options: persistent, transactional and none. • persistent means that your field will be managed and stored in the database on transaction commit. • transactional means that your field will be managed but not stored in the database. Transactional fields values will be saved by JDO when you start your transaction and restored when you roll back your transaction. • none means that your field will not be managed.	persistent, transactional, none
primary-key	Whether the property is part of any primary key (if using application identity).	true, false
null-value	How to treat null values of persistent properties during storage.	exception, default, none

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
default-fetch-group	Whether this property is part of the default fetch group for the class. Defaults to true for non-key fields of primitive types, java.util.Date, java.lang.*, java.math.*, etc.	true, false
embedded	Whether this property should be stored, if possible, as part of the object instead as its own object in the datastore. This defaults to true for primitive types, java.util.Date, java.lang.*, java.math.* etc and false for PersistenceCapable, reference (Object, Interface) and container types.	true, false
serialized	Whether this property should be stored serialised into a single column of the table of the containing object.	true, false
dependent	Whether the property should be used to check for dependent objects, and to delete them when this object is deleted. In other words cascade delete capable.	true, false
mapped-by	The name of the property at the other end of a relationship. Used by 1-1, 1-N, M-N to mark a relation as bidirectional.	
value-strategy	The strategy for populating values to this property. Is typically used for generating primary key values. See the definitions under "datastore-identity".	native | sequence | increment | identity | uuid-string | uuid-hex | auid | max | timestamp | timestamp-value | [other values]
sequence	Name of the sequence to use to generate values, when using a strategy of sequence. Please see also the class extension tags for controlling the sequence.	
recursion-depth	The depth that will be recursed when this property is self-referencing. Should be used alongside FetchPlan.setMaxFetchDepth() to control the objects fetched.	-1, 1, 2, ... (integer)
field-type	Used to specify a more restrictive type than the property definition in the class. This might be required in order to map the field to the datastore. To be portable, specify the name of a single type that is itself able to be mapped to the datastore (e.g. a field of type Object can specify field-type="Integer").	
indexed	Whether the column(s) for this property should be indexed. This is to be specified when defining index information	true | false | unique
table	Table name to use for any join table overriding the default name provided by DataNucleus. This is used either for 1-N relationships with a join table or for Secondary Tables. See also the property name "datanucleus.identifier.case" in the Persistence Properties Guide.	
column	Column name to use for this property (alternative to specifying column sub-elements if only one column).	
delete-action	The foreign-key delete action. This is a shortcut to specifying foreign key information. Please refer to the <foreign-key> element for full details.	cascade | restrict | null | default | none

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

These are attributes within the <extension> tag (jdo/package/class/property/extension).

Attribute	Description	Values
Extension (JDO) Tags		
cascade-persist	JDO defines that when an object is persisted then all fields will also be persisted using "persistence-by-reachability". This extension allows you to turn off the persistence of a field relation.	**true** | false
cascade-update	JDO defines that when an object is updated then all fields containing PersistenceCapable objects will also be updated using "persistence-by-reachability". This extension allows you to turn off the update of a field relation.	**true** | false
cascade-refresh	When calling PersistenceManager.refresh() only fetch plan fields of the passed object will be refreshed. Setting this to true will refresh the fields of related PC objects in this field	true | **false**
allows-null	When the field is a collection by default it will not be allowed to have nulls present but you can allow them by setting this DataNucleus extension tag	true | **false**
insertable	Whether this field should be supplied when inserting into the datastore.	**true** | false
updateable	Whether this field should be supplied when updating the datastore.	**true** | false
adapter-column-name	In some situations DataNucleus will add a special datastore column to a join table so that collections can allow the storage of duplicate elements. This extension allows the specification of the column name to be used. This should be specified within the field at the collection end of the relationship. JDO2 doesnt allow a standard place for such a specification and so is an extension tag.	**INTEGER_IDX**
implementation-classes	Used to define the possible classes implementing this interface/Object field. This is used to limit the possible tables that this is a foreign key to (when this field is specified as an interface/Object in the class). Value should be comma-separated list of fully-qualified class names	
key-implementation-classes	Used to define the possible classes implementing this interface/Object key. This is used to limit the possible tables that this is a foreign key to (when this key is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names	
value-implementation-classes	Used to define the possible classes implementing this interface/Object value. This is used to limit the possible tables that this is a foreign key to (when this value is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
strategy-when-notnull	This is to be used in conjunction with the "value-strategy" attribute. Default JDO2 behaviour when you have a "value-strategy" defined for a field is to always create a strategy value for that field regardless of whether you have set the value of the field yourself. This extension allows you to only apply the strategy if the field is null at persistence. This extension has no effect on primitive field types (which can't be null) and the value-strategy will always be applied to such fields.	**true** | false
relation-discriminator-column	Name of a column to use for discrimination of the relation used by objects stored. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for	**RELATION_DISCRIM**
relation-discriminator-pk	Whether the column added for the discrimination of relations is to be part of the PK when using a join table.	true | **false**
relation-discriminator-value	Value to use in the relation discriminator column for objects of this fields relation. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.	Fully-qualified class name
select-function	Permits to use a function when fetching contents from the database. A ? (question mark) is mandatory to have and will be replaced by the column name when generating the SQL statement. For example to specify a value of *UPPER(?)* will convert to upper case the field value on a datastore that supports that UPPER function.	
insert-function	Permits to use a function when inserting into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of *TRIM(?)* will trim the field value on a datastore that supports that TRIM function.	
update-function	Permits to use a function when updating into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of *FUNC(?)* will perform FUNC() on the field value on a datastore that supports that FUNC function.	
sequence-table-basis	This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the "increment" strategy, and sometimes by "native"). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment	**class** | table
sequence-catalog-name	The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
sequence-schema-name	The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.	
sequence-table-name	The table used to store sequences for use by value generators. See Value Generation.	**SEQUENCE_TABLE**
sequence-name-column-name	The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.	**SEQUENCE_NAME**
sequence-nextval-column-name	The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.	**NEXT_VAL**
key-min-value	The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.	
key-max-value	The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.	
key-start-with	The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.	
key-increment-by	The increment value for use by value generators. Keys will be incremented by this value when being generated. See Value Generation.	
key-database-cache-size	The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	
key-cache-size	The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	
mapping-class	Specifies the mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus mapping class and provide their own mapping class for this field.	Fully-qualified class name

Metadata for fetch-group tag

These are attributes within the <fetch-group> tag (jdo/package/class/fetch-group). This element is used to define fetch groups that are utilised at runtime, and are of particular use with attach/detach. This element can contain fetch-group sub-elements allowing definition of hierarchical groups. It can also contain field elements, defining the fields that are part of this fetch-group.

Attribute	Description	Values
Standard (JDO) Tags		

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
name	Name of the fetch group. Used with the fetch plan of the PersistenceManager.	
post-load	Whether to call jdoPostLoad when the fetch group is invoked.	true | false

Metadata for embedded tag

These are attributes within the <embedded> tag (jdo/package/class/embedded). It is used when this field is a PersistenceCapable and is embedded into the same table as the class.

Attribute	Description	Values
Standard (JDO) Tags		
owner-field	Name of the field in the embedded PersistenceCapable that is the link back to the owning object (if any).	
null-indicator-column	Name of the column to be used for detecting if the embedded object is null.	
null-indicator-value	Value of the null-indicator-column that signifies that the embedded object is null.	

Metadata for key tag

These are attributes within the <key> tag (jdo/package/class/field/key). This element is used to define details for the persistence of a Map.

Attribute	Description	Values
Standard (JDO) Tags		
mapped-by	When the map is formed by a foreign-key, the key can be a field in a value PersistenceCapable class. This attribute defines which field in the value class is used as the key	
column	Name of the column (if only one)	
delete-action	Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information	cascade | restrict | null | default | none
indexed	Whether the key column should be indexed. This is to be specified when defining index information	true | false | unique
unique	Whether the key column should be unique. This is to be specified when defining unique key information	true | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for value tag

These are attributes within the <value> tag (jdo/package/class/field/value). This element is used to define details for the persistence of a Map.

Attribute	Description	Values
mapped-by	When the map is formed by a foreign-key, the value can be a field in a key PersistenceCapable class. This attribute defines which field in the key class is used as the value.	
Standard (JDO) Tags		
column	Name of the column (if only one)	
delete-action	Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information	cascade | restrict | null | default | none
indexed	Whether the value column should be indexed. This is to be specified when defining index information	true | false | unique
unique	Whether the value column should be unique. This is to be specified when defining unique key information	true | false

Metadata for order tag

These are attributes within the <order> tag (jdo/package/class/field/order). This is used to define the column details for the ordering column in a List.

Attribute	Description	Values
Standard (JDO) Tags		
mapped-by	When a List is formed by a foreign-key, the ordering can be a field in the element PersistenceCapable class. This attribute defines which field in the element class is used as the ordering. The field must be of type *int*, *Integer*, *long*, *Long*. DataNucleus will write the index positions to this field (starting at 0 for the first item in the List)	
column	Name of the column to use for ordering.	

These are attributes within the <extension> tag (jdo/package/class/field/order/extension).

Attribute	Description	Values
Extension (JDO) Tags		

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
list-ordering	Used to make the list be an "ordered list" where it has no index column and instead will order the elements by the specified expression upon retrieval. The ordering expression takes names and ASC/DESC and can be a composite	{orderfield [ASC|DESC] [,{orderfield} ASC|DESC]}

Metadata for index tag

These are attributes within the <index> tag (jdo/package/class/field/index). This element is used where a user wishes to add specific indexes to the datastore to provide more efficient access to particular fields.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the index in the datastore	
unique	Whether the index is unique	true	false
column	Name of the column to use (alternative to specifying it as a sub-element).	

These are attributes within the <extension> tag (jdo/package/class/field/index/extension).

Attribute	Description	Values
Extension (JDO) Tags		
extended-setting	Additional settings to the index. This extension is used to set database proprietary settings.	

Metadata for foreign-key tag

These are attributes within the <foreign-key> tag (jdo/package/class/field/foreign-key). This is used where the user wishes to define the behaviour of the foreign keys added due to the relationships in the object model. This is to be read in conjunction with foreign-key guide

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the foreign key in the datastore	
deferred	Whether the constraints are initially deferred.	true	false
delete-action	Action to be performed when the owner object is deleted.	cascade	restrict	null	default

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
update-action	Action to be performed when the owner object is updated.	cascade | restrict | null | default

Metadata for unique tag

These are attributes within the <unique> tag (jdo/package/class/unique, jdo/package/class/field/unique). This element is used where a user wishes to add specific unique constraints to the datastore to provide more control over particular fields.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the constraint in the datastore	
column	Name of the column to use (alternative to specifying it as a sub-element).	

Metadata for column tag

These are attributes within the <column> tag (*/column). This is used to define the details of a column in the datastore, and so can be used to match to an existing datastore schema.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the column in the datastore. See also the property name "datanucleus.identifier.case" in the Persistence Properties Guide.	
length	Length of the column in the datastore (for character types), or the precision of the column in the datastore (for floating point field types).	positive integer
scale	Scale of the column in the datastore (for floating point field types).	positive integer
jdbc-type	JDBC Type to use for this column in the datastore when the default value is not satisfactory. Please refer to JDBC for the valid types. Not all of these types are supported for all RDBMS mappings.	Valid JDBC Type (CHAR, VARCHAR, LONGVARCHAR, NUMERIC, DECIMAL, BIT, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP, BLOB, BOOLEAN, CLOB, DATALINK)
sql-type	SQL Type to use for this column in the datastore. This should not usually be necessary since the specification of JDBC type together with length/scale will likely define it.	Valid SQL Type (e.g VARCHAR, CHAR, NUMERIC etc)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
allows-null	Whether the column in the datastore table should allow nulls or not. The default is "false" for primitives, and "true" otherwise.	true | false
default-value	Default value to use for this column when creating the table.	Default value expression
target	Declares the name of the primary key column for the referenced table. For columns contained in join elements, this is the name of the primary key column in the primary table. For columns contained in field, element, key, value, or array elements, this is the name of the primary key column of the primary table of the other side of the relationship.	target column name
target-field	Declares the name of the primary key field for the referenced class. For columns contained in join elements, this is the name of the primary key field in the base class. For columns contained in field, element, key, value, or array elements, this is the name of the primary key field of the base class of the other side of the relationship.	target field name
insert-value	Value to use for this column when it has no field in the class and an object is being inserted	Insert value

These are attributes within the <extension> tag (*/column/extension).

Attribute	Description	Values
Extension (JDO) Tags		
datastore-mapping-class	Specifies the datastore mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus datastore mapping class and provide their own mapping class for this field based on the database data type. This datastore mapping class must be available for the DataNucleus PersistenceManagerFactory classpath.	Fully-qualified class name
enum-check-constraint	Specifies that a CHECK constraint for this column must be generated based on the values of a java.lang.Enum type. e.g. enum Color (RED, GREEN, BLUE) where its name is persisted a CHECK constraint is defined as *CHECK "COLUMN" IN ('RED', 'GREEN', 'BLUE').*	true | false

Metadata for join tag

These are attributes within the <join> tag (jdo/package/class/field/join). This element is added when the field has a mapping to a "join" table (as part of a 1-N relationship). It is also used to specify overriding of details in an inheritance tree where the primary key columns are shared up the hierarchy. A further use (when specified under the <class> element) is for specifying the column details for joining to a Secondary Table.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
Standard (JDO) Tags		
column	Name of the column used to join to the PK of the primary table (when only one column used). Used in Secondary Tables.	
table	Table name used when joining the PK of a FCO class table to a secondary table. See Secondary Tables.	
delete-action	Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information	cascade | restrict | null | default | none
indexed	Whether the join table owner column should be indexed. This is to be specified when defining index information	true | false | unique
unique	Whether the join table owner column should be unique. This is to be specified when defining unique key information	true | false
outer	Whether to use an outer join here. This is of particular relevance to secondary tables	true | **false**

These are attributes within the <extension> tag (jdo/package/class/field/join/extension). These are for controlling the join table.

Attribute	Description	Values
Extension (JDO) Tags		
primary-key	This parameter defines if the join table will be assigned a primary key. The default is true since it is considered a best practice to have primary keys on all tables. This allows the option of turning it off.	**true** | false

Metadata for element tag

These are attributes within the <element> tag (jdo/package/class/field/element). This element is added when the field has a mapping to a "element" (as part of a 1-N relationship).

Attribute	Description	Values
Standard (JDO) Tags		
mapped-by	The name of the field at the other ("N") end of a relationship when this field is the "1" side of a 1-N relationship (for FK relationships). This performs the same function as specifying "mapped-by" on the <field> element.	
column	Name of the column (alternative to specifying it as a sub-element).	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
delete-action	Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information	cascade | restrict | null | default | none
indexed	Whether the element column should be indexed. This is to be specified when defining index information	true | false | unique
unique	Whether the element column should be unique. This is to be specified when defining unique key information	true | false

Metadata for collection tag

These are attributes within the <collection> tag (jdo/package/class/field/collection). This is used to define the persistence of a Collection.

Attribute	Description	Values
Standard (JDO) Tags		
element-type	The type of element stored in this Collection or array (fully qualified class). This is not required when the field is an array. It is also not required when the Collection is defined using JDK 1.5 generics.	
embedded-element	Whether the elements of a collection or array-valued persistent field should be stored embedded or as first-class objects. It's a hint for the JDO implementation to store, if possible, the elements of the collection as part of the it instead of as their own instances in the datastore. See the <embedded> element for details on how to define the field mappings for the embedded element.	true, false
dependent-element	Whether the elements of the collection are to be considered dependent on the owner object.	true, false
serialized-element	Whether the elements of a collection or array-valued persistent field should be stored serialised into a single column of the join table (where used).	true, false

These are attributes within the <extension> tag (jdo/package/class/field/collection/extension).

Attribute	Description	Values
Extension (JDO) Tags		
cache	Whether this SCO collection will be cached by DataNucleus or whether every access of the collection will go through to the datastore. See also "datanucleus.cache.collections" in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory	**true** | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
cache-lazy-loading	Whether objects from this SCO collection will be lazy loaded (loaded when required) or whether they should be loaded at initialisation. See also "datanucleus.cache.collections.lazy" in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory	**true** | false
comparator-name	Defines the name of the comparator to use with SortedSet, TreeSet collections. The specified name is the name of the comparator class, which must have a default constructor. This extension is only used by SortedSet, TreeSet fields.	Fully-qualified class name

Metadata for map tag

These are attributes within the <map> tag (jdo/package/class/field/map). This is used to define the persistence of a Map.

Attribute	Description	Values
Standard (JDO) Tags		
key-type	The type of key stored in this Map (fully qualified class). This is not required when the Map is defined using JDK 1.5 generics.	
embedded-key	Whether the elements of a Map key field should be stored embedded or as first-class objects.	true, false
value-type	The type of value stored in this Map (fully qualified class). This is not required when the Map is defined using JDK 1.5 generics.	
embedded-value	Whether the elements of a Map value field should be stored embedded or as first-class objects.	true, false
dependent-key	Whether the keys of the map are to be considered dependent on the owner object.	true, false
dependent-value	Whether the value of the map are to be considered dependent on the owner object.	true, false
serialized-key	Whether the keys of a map-valued persistent field should be stored serialised into a single column of the join table (where used).	true, false
serialized-value	Whether the values of a map-valued persistent field should be stored serialised into a single column of the join table (where used).	true, false

These are attributes within the <extension> tag (jdo/package/class/field/map/extension).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
Extension (JDO) Tags		
cache	Whether this SCO map will be cached by DataNucleus or whether every access of the map will go through to the datastore. See also "datanucleus.cache.collections" in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory	**true** | false
cache-lazy-loading	Whether objects from this SCO map will be lazy loaded (loaded when required) or whether they should be loaded at initialisation. See also "datanucleus.cache.collections.lazy" in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory	**true** | false
comparator-name	Defines the name of the comparator to use with SortedMap, TreeMap maps. The specified name is the name of the comparator class, which must have a default constructor. This extension is only used by SortedMap, TreeMap fields.	Fully-qualified class name

Metadata for array tag

This is used to define the persistence of an array. DataNucleus provides support for many types of arrays, either serialised into a single column, using a join table, or via a foreign-key (for arrays of PC objects).

Attribute	Description	Values
Standard (JDO) Tags		
embedded-element	Whether the array elements should be stored embedded (default = true for primitives, wrappers etc and false for PersistenceCapable objects).	true, false
serialized-element	Whether the array elements should be stored serialised into a single column in the join table.	true, false
dependent-element	Whether the elements of the array are to be considered dependent on the owner object.	true, false

Metadata for sequence tag

These are attributes within the <sequence> tag. This is used to denote a JDO datastore sequence.

Attribute	Description	Values
Standard (JDO) Tags		
name	Symbolic name for the sequence for this package	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
datastore-sequence	Name of the sequence in the datastore	
factory-class	Factory class for creating the sequence. Please refer to the Sequence guide	
strategy	Strategy to use for application of this sequence.	nontransactional | contiguous | noncontiguous

These are attributes within the <extension> tag (jdo/package/class/sequence/extension). These are for controlling the datastore sequences created by DataNucleus. Please refer to the documentation for the value generator being used for applicability

Attribute	Description	Values
Extension (JDO) Tags		
sequence-catalog-name	The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.	
sequence-schema-name	The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.	
sequence-table-name	The table used to store sequences for use by value generators. See Value Generation.	**SEQUENCE_TABLE**
sequence-name-column-name	The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.	**SEQUENCE_NAME**
sequence-nextval-column-name	The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.	**NEXT_VAL**
key-min-value	The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.	
key-max-value	The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.	
key-start-with	The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.	
key-increment-by	The increment value for use by value generators. Keys will be incremented by this value when being generated. See Value Generation.	
key-database-cache-size	The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
key-cache-size	The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.	

Metadata for fetch-plan tag

These are attributes within the <fetch-plan> tag (jdo/fetch-plan). This element is used to define fetch plans that are utilised at runtime, and are of particular use with queries. This element contains fetch-group sub-elements.

Attribute	Description	Values
Standard (JDO) Tags		
name	Name of the fetch plan.	
maxFetchDepth	Max depth to fetch with this fetch plan	1
fetchSize	Size to fetch with this fetch plan (for use with query result sets	0

Metadata for class extension tag

These are attributes within the <extension> tag (jdo/package/class/extension). These are for controlling the class definition

Attribute	Description	Values
Extension (JDO) Tags		
requires-table	This is for use with a "nondurable" identity case and specifies whether the class requires a table/view in the datastore.	true | false
ddl-definition	Definition of the TABLE SCHEMA to be used by the class.	true | false
ddl-imports	Classes imported resolve macro identifiers in the definition of a RDBMS Table.	
mysql-engine-type	"Engine Type" to use when creating the table for this class in MySQL. Refer to the MySQL documentation for ENGINE type (e.g INNODB, MEMORY, ISAM)	
view-definition	Definition of the VIEW to be used by the class. Please refer to the RDBMS Views Guide for details. If your view already exists, then specify this as " " and have the autoStart flags set to false.	
view-imports	Classes imported resolve macro identifiers in the definition of a RDBMS View. Please refer to the RDBMS Views Guide for details.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for extension tag

These are attributes within the <extension> tag. This is used to denote a DataNucleus extension to JDO.

Attribute	Description	Values
Standard (JDO) Tags		
vendor-name	Name of the vendor. For DataNucleus we use the name "datanucleus" (lowercase).	
key	Key of the extension property	
value	Value of the extension property	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.10.2 JDO Annotations

JDO Annotations

One of the things that JDK 1.5 provides that can be of some use is annotations. JDO 2.1 introduces support for annotations to the JDO standard. When selecting to use annotations please bear in mind the following :-

- You must be using **JDK 1.5 or above**.
- You must have **JDO 2.1 or later** in your CLASSPATH since this provides the annotations
- Annotations should really only be used for attributes of persistence that you won't be changing at deployment. Things such as table and column names shouldn't really be specified using annotations although it is permitted. Instead it would be better to put such information in an ORM MetaData file.
- Annotations can be added in two places - for the class as a whole, or for a field in particular.
- You can annotate fields or getters with field-level information. If you annotate fields then the fields are processed for persistence. If you annotate the methods (getters) then the methods (properties) are processed for persistence.
- Annotations are prefixed by the @ symbol and can take properties (in brackets after the name, comma-separated)

Annotations supported by DataNucleus are shown below. The annotations/attributes coloured in pink are ORM and really should be placed in XML rather than directly in the class using annotations.

Annotation	Class/Field/Method	Description
@PersistenceCapable	Class	Specifies that the class/interface is persistent. In the case of an interface this would utilise JDO2's "persistent-interface" capabilities
@PersistenceAware	Class	Specifies that the class is not persistent but needs to be able to access fields of persistent classes
@Cacheable	Class	Specifies whether this class can be cached in a Level 2 cache or not.
@EmbeddedOnly	Class	Specifies that the class is persistent and can only be persisted embedded in another persistent class
@DatastoreIdentity	Class	Specifies the details for generating datastore-identity for this class
@Version	Class	Specifies any versioning process for objects of this class
@FetchPlans	Class	Defines a series of fetch plans
@FetchPlan	Class	Defines a fetch plan
@FetchGroups	Class	Defines a series of fetch groups for this class
@FetchGroup	Class	Defines a fetch group for this class

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Annotation	Class/Field/Method	Description
@Sequence	Class	Defines a sequence for use by this class
@Queries	Class	Defines a series of named queries for this class
@Query	Class	Defines a named query for this class
@Inheritance	Class	Specifies the inheritance model for persisting this class
@Discriminator	Class	Specifies any discriminator for this class to be used for determining object types
@PrimaryKey	Class	ORM : Defines the primary key constraint for this class
@Indices	Class	ORM : Defines a series of indices for this class
@Index	Class	ORM : Defines an index for the class as a whole (typically a composite index)
@Uniques	Class	ORM : Defines a series of unique constraints for this class
@Unique	Class	ORM : Defines a unique constraint for the class as a whole (typically a composite)
@ForeignKeys	Class	ORM : Defines a series of foreign-keys (typically for non-mapped columns/tables)
@ForeignKey	Class	ORM : Defines a foreign-key for the class as a whole (typically for non-mapped columns/tables)
@Joins	Class	ORM : Defines a series of joins to secondary tables from this table
@Join	Class	ORM : Defines a join to a secondary table from this table
@Columns	Class	ORM : Defines a series of columns that dont have associated fields ("unmapped columns")
@Persistent	Field/Method	Defines the persistence for a field/property of the class
@Serialized	Field/Method	Defines this field as being stored serialised
@NotPersistent	Field/Method	Defines this field as being not persisted
@Transactional	Field/Method	Defines this field as being transactional (not persisted, but managed)
@Cacheable	Field/Method	Specifies whether this field/property can be cached in a Level 2 cache or not.
@PrimaryKey	Field/Method	Defines this field as being (part of) the primary key
@Element	Field/Method	Defines the details of elements of an array/collection stored in this field
@Key	Field/Method	Defines the details of keys of a map stored in this field

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Annotation	Class/Field/Method	Description
@Value	Field/Method	Defines the details of values of a map stored in this field
@Order	Field/Method	ORM : Defines the details of ordering of an array/collection stored in this field
@Join	Field/Method	ORM : Defines the join to a join table for a collection/array/map
@Embedded	Field/Method	ORM : Defines that this field is embedded and how it is embedded
@Columns	Field/Method	ORM : Defines a series of columns where a field is persisted
@Column	Field/Method	ORM : Defines a column where a field is persisted
@Index	Field/Method	ORM : Defines an index for the field
@Unique	Field/Method	ORM : Defines a unique constraint for the field
@ForeignKey	Field/Method	ORM : Defines a foreign key for the field
@Extensions	Class/Field/Method	Defines a series of JDO extensions
@Extension	Class/Field/Method	Defines a JDO extension

@PersistenceCapable

This annotation is used when you want to mark a class as persistent. It equates to the <class> MetaData element (though with only some of its attributes). Specified on the **class**.

Attribute	Type	Description	Default
requiresExtent	String	Whether an extent is required for this class	true
embeddedOnly	String	Whether objects of this class can only be stored embedded in other objects	false
detachable	String	Whether objects of this class can be detached	false
identityType	IdentityType	Type of identity (APPLICATION, DATASTORE, NONDURABLE)	DATASTORE
objectIdClass	Class	Object-id class	
table	String	ORM : Name of the table where this class is persisted	
catalog	String	ORM : Name of the catalog where this table is persisted	
schema	String	ORM : Name of the schema where this table is persisted	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
cacheable	String	Whether the class can be L2 cached. **From JDO2.2**	**true** | false
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable(identityType=IdentityType.APPLICATION)
public class MyClass
{
    ...
}
```

@PersistenceAware

This annotation is used when you want to mark a class as being used in persistence but not being persistable. That is "persistence-aware" in JDO2 terminology. It has no attributes. Specified on the **class**.

```
@PersistenceAware
public class MyClass
{
    ...
}
```

@Cacheable

This annotation is a shortcut for @PersistenceCapable(cacheable={value}) specifying whether the class can be cached in a Level 2 cache. **This is new in JDO 2.2** Specified on the **class**. The default

Attribute	Type	Description	Default
value	String	Whether the class is cacheable	**true** | false

```
@Cacheable("false")
public class MyClass
{
    ...
}
```

@EmbeddedOnly

This annotation is a shortcut for @PersistenceCapable(embeddedOnly="true") meaning that the class can only be persisted embedded into another class. It has no attributes. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@EmbeddedOnly
public class MyClass
{
    ...
}
```

@Inheritance

Annotation used to define the inheritance for a class. Specified on the **class**.

Attribute	Type	Description	Default
strategy	InheritanceStrategy	The inheritance strategy (NEW_TABLE, SUBCLASS_TABLE, SUPERCLASS_TABLE)	
customStrategy	String	Name of a custom inheritance strategy (DataNucleus supports "complete-table")	

```
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
public class MyClass
{
    ...
}
```

@Discriminator

Annotation used to define a discriminator to be stored with instances of this class and is used to determine the types of the objects being stored. Specified on the **class**.

Attribute	Type	Description	Default
strategy	DiscriminatorStrategy	The discriminator strategy (VALUE_MAP, CLASS_NAME, NONE)	
value	String	Value to use for instances of this type when using strategy of VALUE_MAP	
column	String	ORM : Name of the column to use to store the discriminator	
indexed	String	ORM : Whether the discriminator column is to be indexed	
columns	Column[]	ORM : Column definitions used for storing the discriminator	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
@Discriminator(strategy=DiscriminatorStrategy.CLASS_NAME)
public class MyClass
{
    ...
}
```

@DatastoreIdentity

Annotation used to define the identity when using datastore-identity for the class. Specified on the **class**.

Attribute	Type	Description	Default
strategy	IdGeneratorStrategy	The inheritance strategy (NATIVE, SEQUENCE, IDENTITY, INCREMENT, UUIDSTRING, UUIDHEX)	
customStrategy	String	Name of a custom id generation strategy (e.g "max", "auid"). This overrides the value of "strategy"	
sequence	String	Name of the sequence to use (when using SEQUENCE strategy) - refer to @Sequence	
column	String	ORM : Name of the column for the datastore identity	
columns	Column[]	ORM : Column definition for the column(s) for the datastore identity	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
@DatastoreIdentity(strategy=IdGeneratorStrategy.INCREMENT)
public class MyClass
{
    ...
}
```

@Version

Annotation used to define the versioning details for use with optimistic transactions. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
strategy	VersionStrategy	The version strategy (NONE, STATE_IMAGE, DATE_TIME, VERSION_NUMBER)	
indexed	String	Whether the version column(s) is indexed	
column	String	ORM : Name of the column for the version	
columns	Column[]	ORM : Column definition for the column(s) for the version	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
@Version(strategy=VersionStrategy.VERSION_NUMBER)
public class MyClass
{
    ...
}
```

@PrimaryKey

Annotation used to define the primary key constraint for a class. Maps across to the <primary-key> MetaData element. Specified on the **class**.

Attribute	Type	Description	Default
name	String	ORM : Name of the primary key constraint	
column	String	ORM : Name of the column for this key	
columns	Column[]	ORM : Column definition for the column(s) of this key	

```
@PersistenceCapable
@PrimaryKey(name="MYCLASS_PK")
public class MyClass
{
    ...
}
```

@FetchPlans

Annotation used to define a set of fetch plans. Specified on the **class**. Used by named queries

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
value	FetchPlan[]	Array of fetch plans - see @FetchPlan annotation	

```
@PersistenceCapable
@FetchPlans({@FetchPlan(name="plan_3", maxFetchDepth=3, fetchGroups={"group1",
"group4"}),
             @FetchPlan(name="plan_4", maxFetchDepth=2, fetchGroups={"group1",
"group2"})})
public class MyClass
{
    ...
}
```

@FetchPlan

Annotation used to define a fetch plan Is equivalent to the <fetch-plan> metadata element. Specified on the **class**. Used by named queries

Attribute	Type	Description	Default
name	String	Name of the FetchPlan	
maxFetchDepth	int	Maximum fetch depth	1
fetchSize	int	Size hint for fetching query result sets	0
fetchGroups	String[]	Names of the fetch groups included in this FetchPlan.	

```
@PersistenceCapable
@FetchPlan(name="plan_3", maxFetchDepth=3, fetchGroups={"group1", "group4"})
public class MyClass
{
    ...
}
```

@FetchGroups

Annotation used to define a set of fetch groups for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	FetchGroup[]	Array of fetch groups - see @FetchGroup annotation	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@PersistenceCapable
@FetchGroups({@FetchGroup(name="one_two", members={@Persistent(name="field1"),
@Persistent(name="field2")}),
              @FetchGroup(name="three", members={@Persistent(name="field3")})})
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;

    @Persistent
    String field3;
    ...
}
```

@FetchGroup

Annotation used to define a fetch group. Is equivalent to the <fetch-group> metadata element. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the fetch group	
postLoad	String	Whether to call jdoPostLoad after loading this fetch group	
members	Persistent[]	Definitions of the fields/properties to include in this fetch group	

```
@PersistenceCapable
@FetchGroup(name="one_two", members={@Persistent(name="field1"),
@Persistent(name="field2")})
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;
    ...
}
```

@Sequence

Annotation used to define a sequence generator. Is equivalent to the <sequence> metadata element. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	Name of the sequence	
strategy	SequenceStrategy	Strategy for the sequence (NONTRANSACTIONAL, CONTIGUOUS, NONCONTIGUOUS)	
datastoreSequence	String	Name of a datastore sequence that this maps to	
factoryClass	Class	Factory class to use to generate the sequence	
extensions	Extension[]	Vendor extensions	

@Queries

Annotation used to define a set of named queries for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	Query[]	Array of queries - see @Query annotation	

```
@PersistenceCapable
@Queries({@Query(name="PeopleCalledSmith", language="JDOQL",
                 value="SELECT FROM org.datanucleus.samples.Person WHERE surname ==
\"Smith\""),
          @Query(name="PeopleCalledJones", language="JDOQL",
                 value="SELECT FROM org.datanucleus.samples.Person WHERE surname ==
\"Jones\"")})
public class Person
{
    @Persistent
    String surname;

    ...
}
```

@Query

Annotation used to define a named query. Is equivalent to the <query> metadata element. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the query	
value	String	The query string itself	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
language	String	Language of the query (JDOQL, SQL, ...)	JDOQL
unmodifiable	String	Whether the query is not modifiable at runtime	
unique	String	Whether the query returns unique results (for SQL queries only)	
resultClass	Class	Result class to use (for SQL queries only)	
fetchPlan	String	Name of a named FetchPlan to use with this query	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
@Query(name="PeopleCalledSmith", language="JDOQL",
       value="SELECT FROM org.datanucleus.samples.Person WHERE surname ==
\"Smith\"")
public class Person
{
    @Persistent
    String surname;

    ...
}
```

@Indices

Annotation used to define a set of indices for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	Index[]	Array of indices - see @Index annotation	

```
@PersistenceCapable
@Indices({@Index(name="MYINDEX_1", members={"field1","field2"}),
@Index(name="MYINDEX_2", members={"field3"})})
public class Person
{
    ...
}
```

@Index

Annotation used to define an index for the class as a whole typically being a composite index across

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

multiple columns or fields/properties. Is equivalent to the <index> metadata element when specified under class. Specified on the **class**.

Attribute	Type	Description	Default
name	String	ORM : Name of the index	
table	String	ORM : Name of the table for the index	
unique	String	ORM : Whether the index is unique	
members	String[]	ORM : Names of the fields/properties that make up this index	
columns	Column[]	ORM : Columns that make up this index	

```
@PersistenceCapable
@Index(name="MY_COMPOSITE_IDX", members={"field1", "field2"})
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;

    ...
}
```

@Uniques

Annotation used to define a set of unique constraints for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	Unique[]	Array of constraints - see @Unique annotation	

```
@PersistenceCapable
@Uniques({@Unique(name="MYCONST_1", members={"field1","field2"}),
@Unique(name="MYCONST_2", members={"field3"})})
public class Person
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@Unique

Annotation used to define a unique constraints for the class as a whole typically being a composite constraint across multiple columns or fields/properties. Is equivalent to the <unique> metadata element when specified under class. Specified on the **class**.

Attribute	Type	Description	Default
name	String	ORM : Name of the constraint	
table	String	ORM : Name of the table for the constraint	
deferred	String	ORM : Whether the constraint is deferred	
members	String[]	ORM : Names of the fields/properties that make up this constraint	
columns	Column[]	ORM : Columns that make up this constraint	

```
@PersistenceCapable
@Unique(name="MY_COMPOSITE_IDX", members={"field1", "field2"})
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;

    ...
}
```

@ForeignKeys

Annotation used to define a set of foreign-key constraints for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	ForeignKey[]	Array of FK constraints - see @ForeignKey annotation	

@ForeignKey

Annotation used to define a foreign-key constraint for the class. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	ORM : Name of the constraint	
table	String	ORM : Name of the table that the FK is to	
deferred	String	ORM : Whether the constraint is deferred	
unique	String	ORM : Whether the constraint is unique	
deleteAction	ForeignKeyAction	ORM : Action to apply to the FK to be used on deleting	ForeignKeyAction.RESTRICT
updateAction	ForeignKeyAction	ORM : Action to apply to the FK to be used on updating	ForeignKeyAction.RESTRICT
members	String[]	ORM : Names of the fields/properties that compose this FK.	
columns	Column[]	ORM : Columns that compose this FK.	

@Joins

Annotation used to define a set of joins (to secondary tables) for a class. Specified on the **class**.

Attribute	Type	Description	Default
value	Join[]	Array of joins - see @Join annotation	

```
@PersistenceCapable
@Joins({@Join(table="MY_OTHER_TABLE", column="MY_PK_COL"),
        @Join(table="MY_SECOND_TABLE", column="MY_PK_COL")})
public class MyClass
{
    @Persistent(table="MY_OTHER_TABLE")
    String myField;

    @Persistent(table="MY_SECOND_TABLE")
    String myField2;
    ...
}
```

@Join

Annotation used to specify a join for a secondary table. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
table	String	ORM : Table name used when joining the PK of a FCO class table to a secondary table.	
column	String	ORM : Name of the column used to join to the PK of the primary table (when only one column used)	
outer	String	ORM : Whether to use an outer join when retrieving fields/properties stored in the secondary table	
columns	Column[]	ORM : Name of the colums used to join to the PK of the primary table (when multiple columns used)	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable(name="MYTABLE")
@Join(table="MY_OTHER_TABLE", column="MY_PK_COL")
public class MyClass
{
    @Persistent(name="MY_OTHER_TABLE")
    String myField;
    ...
}
```

@Columns

Annotation used to define the columns which have no associated field in the class. User should specify a minimum of @Column "name", "jdbcType", and "insertValue". Specified on the **class**.

Attribute	Type	Description	Default
value	Column[]	Array of columns - see @Column annotation	

```
@PersistenceCapable
@Columns(@Column(name="MY_OTHER_COL", jdbcType="VARCHAR", insertValue="N/A")
public class MyClass
{
    ...
}
```

@Persistent

Annotation used to define the fields/properties to be persisted. Is equivalent to the <field> and

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

<property> metadata elements. Specified on the **field/method**.

Attribute	Type	Description	Default
persistenceModifier	PersistenceModifier	Whether the field is persistent (PERSISTENT, TRANSACTIONAL, NONE)	[depends on field type]
defaultFetchGroup	String	Whether the field is part of the DFG	
nullValue	NullValue	Required behaviour when inserting a null value for this field (NONE, EXCEPTION, DEFAULT).	NONE
embedded	String	Whether this field is embedded	
embeddedElement	String	Whether the element stored in this field/property is embedded	
embeddedKey	String	Whether the key stored in this field/property is embedded	
embeddedValue	String	Whether the value stored in this field/property is embedded	
serialized	String	Whether this field is serialised	
serializedElement	String	Whether the element stored in this field/property is serialised	
serializedKey	String	Whether the key stored in this field/property is serialised	
serializedValue	String	Whether the value stored in this field/property is serialised	
dependent	String	Whether this field is dependent, deleting the related object when deleting this object	
dependentElement	String	Whether the element stored in this field/property is dependent	
dependentKey	String	Whether the key stored in this field/property is dependent	
dependentValue	String	Whether the value stored in this field/property is dependent	
primaryKey	String	Whether this field is (part of) the primary key	false
valueStrategy	IdGeneratorStrategy	Strategy to use when generating values for the field (NATIVE, SEQUENCE, IDENTITY, INCREMENT, UUIDSTRING, UUIDHEX)	
customValueStrategy	String	Name of a custom id generation strategy (e.g "max", "auid"). This overrides the value of "valueStrategy"	
sequence	String	Name of the sequence when using valueStrategy of SEQUENCE - refer to @Sequence	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
loadFetchGroup	String	Whether to load the fetch group	
types	Class[]	Type(s) of field (when using interfaces/reference types). DataNucleus currently only supports the first value although in the future it is hoped to support multiple.	
mappedBy	String	Field in other class when the relation is bidirectional to signify the owner of the relation	
recursionDepth	int	Recursion depth for this field when fetching	1
table	String	ORM : Name of the table where this field is persisted. If this field is a collection/map/array then the table refers to a join table, otherwise this refers to a secondary table.	
name	String	Name of the field when defining an embedded field.	
columns	Column[]	ORM : Column definition(s) for the columns into which this field is persisted. This is only typically used when specifying columns of a field of an embedded class.	
cacheable	String	Whether the field/property can be L2 cached. **From JDO2.2**	**true** | false
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Persistent(primaryKey="true")
    String myField;
    ...
}
```

@Serialized

This annotation is a shortcut for @Persistent(serialized="true") meaning that the field is stored serialized. It has no attributes. Specified on the **field/method**.

```
@PersistenceCapable
public class MyClass
{
    @Serialized
    Object myField;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@NotPersistent

This annotation is a shortcut for @Persistent(persistenceModifier=PersistenceModifier.NONE) meaning that the field/property is not persisted. It has no attributes. Specified on the **field/method**.

```
@PersistenceCapable
public class MyClass
{
    @NotPersistent
    String myOtherField;
    ...
}
```

@Transactional

This annotation is a shortcut for @Persistent(persistenceModifier=PersistenceModifier.TRANSACTIONAL) meaning that the field/property is not persisted yet managed. It has no attributes. Specified on the **field/method**.

```
@PersistenceCapable
public class MyClass
{
    @Transactional
    String myOtherField;
    ...
}
```

@Cacheable

This annotation is a shortcut for @Persistent(cacheable={value}) specifying whether the field/property can be cached in a Level 2 cache. **This is new in JDO 2.2** Specified on the **field/property**. The default

Attribute	Type	Description	Default
value	String	Whether the field/property is cacheable	**true** | false

```
public class MyClass
{
    @Cacheable("false")
    Collection elements;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@PrimaryKey

This annotation is a shortcut for @Persistent(primaryKey="true") meaning that the field/property is part of the primary key for the class. No attributes are needed when specified like this. Specified on the **field/method**.

```
@PersistenceCapable
public class MyClass
{
    @PrimaryKey
    String myOtherField;
    ...
}
```

@Element

Annotation used to define the element for any collection/array to be persisted. Maps across to the <collection>, <array> and <element> MetaData elements. Specified on the **field/method**.

Attribute	Type	Description	Default
types	Class[]	Type(s) of element. While the attribute allows multiple values DataNucleus currently only supports the first type value	When using an array is not needed. When using a collection will be taken from the collection definition if using generics, otherwise must be specified.
embedded	String	Whether the element is embedded into a join table	
serialized	String	Whether the element is serialised into the join table	
dependent	String	Whether the element objects are dependent when deleting the owner collection/array	
mappedBy	String	Field in the element class that represents this object (when the relation is bidirectional)	
embeddedMapping	Embedded[]	Definition of any embedding of the (persistable) element. Only 1 "Embedded" should be provided	
table	String	ORM : Name of the table for this element	
column	String	ORM : Name of the column for this element	
foreignKey	String	ORM : Name of any foreign-key constraint to add	
generateForeignKey	String	ORM : Whether to generate a FK constraint for the element (when not specifying the name)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
deleteAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this element for action upon deletion	
updateAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this element for action upon update	
index	String	ORM : Name of any index constraint to add	
indexed	String	ORM : Whether this element column is indexed	
unique	String	ORM : Whether this element column is unique	
uniqueKey	String	ORM : Name of any unique key constraint to add	
columns	Column[]	ORM : Column definition for the column(s) of this element	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Element(types=org.datanucleus.samples.MyElementClass.class, dependent="true")
    Collection myField;
    ...
}
```

@Order

Annotation used to define the ordering of an order-based Collection/array to be persisted. Maps across to the <order> MetaData element. Specified on the **field/method**.

Attribute	Type	Description	Default
mappedBy	String	ORM : Field in the element class that represents the ordering of the collection/array	
column	String	ORM : Name of the column for this order	
columns	Column[]	ORM : Column definition for the column(s) of this order	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    @Element(types=org.datanucleus.samples.MyElementClass.class, dependent="true")
    @Order(column="ORDER_IDX")
    Collection myField;
    ...
}
```

@Key

Annotation used to define the key for any map to be persisted. Maps across to the <map> and <key> MetaData elements. Specified on the **field/method**.

Attribute	Type	Description	Default
types	Class[]	Type(s) of key. While the attribute allows multiple values DataNucleus currently only supports the first type value	When using generics will be taken from the Map definition, otherwise must be specified
embedded	String	Whether the key is embedded into a join table	
serialized	String	Whether the key is serialised into the join table	
dependent	String	Whether the key objects are dependent when deleting the owner map	
mappedBy	String	Used to specify the field in the value class where the key is stored (optional).	
embeddedMapping	Embedded[]	Definition of any embedding of the (persistable) key. Only 1 "Embedded" should be provided	
table	String	ORM : Name of the table for this key	
column	String	ORM : Name of the column for this key	
foreignKey	String	ORM : Name of any foreign-key constraint to add	
generateForeignKey	String	ORM : Whether to generate a FK constraint for the key (when not specifying the name)	
deleteAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this key for action upon deletion	
updateAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this key for action upon update	
index	String	ORM : Name of any index constraint to add	
indexed	String	ORM : Whether this key column is indexed	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
uniqueKey	String	ORM : Name of any unique key constraint to add	
unique	String	ORM : Whether this key column is unique	
columns	Column[]	ORM : Column definition for the column(s) of this key	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Key(types=java.lang.String.class)
    Map myField;
    ...
}
```

@Value

Annotation used to define the value for any map to be persisted. Maps across to the <map> and <value> MetaData elements. Specified on the **field/method**.

Attribute	Type	Description	Default
types	Class[]	Type(s) of value. While the attribute allows multiple values DataNucleus currently only supports the first type value	When using generics will be taken from the Map definition, otherwise must be specified
embedded	String	Whether the value is embedded into a join table	
serialized	String	Whether the value is serialised into the join table	
dependent	String	Whether the value objects are dependent when deleting the owner map	
mappedBy	String	Used to specify the field in the key class where the value is stored (optional).	
embeddedMapping	Embedded[]	Definition of any embedding of the (persistable) value. Only 1 "Embedded" should be provided	
table	String	ORM : Name of the table for this value	
column	String	ORM : Name of the column for this value	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
foreignKey	String	ORM : Name of any foreign-key constraint to add	
deleteAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this value for action upon deletion	
generateForeignKey	String	ORM : Whether to generate a FK constraint for the value (when not specifying the name)	
updateAction	ForeignKeyAction	ORM : Action to be applied to the foreign key for this value for action upon update	
index	String	ORM : Name of any index constraint to add	
indexed	String	ORM : Whether this value column is indexed	
uniqueKey	String	ORM : Name of any unique key constraint to add	
unique	String	ORM : Whether this value column is unique	
columns	Column[]	ORM : Column definition for the column(s) of this value	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Key(types=java.lang.String.class)
    @Value(types=org.datanucleus.samples.MyValueClass.class, dependent="true")
    Map myField;
    ...
}
```

@Join

Annotation used to specify a join to a join table for a collection/array/map. Specified on the **field/method**.

Attribute	Type	Description	Default
table	String	ORM : Name of the table	
column	String	ORM : Name of the column to join our PK to in the join table (when only one column used)	
primaryKey	String	ORM : Name of any primary key constraint to add for the join table	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
generatePrimaryKey	String	ORM : Whether to generate a PK constraint on the join table (when not specifying the name)	
foreignKey	String	ORM : Name of any foreign-key constraint to add	
generateForeignKey	String	ORM : Whether to generate a FK constraint on the join table (when not specifying the name)	
index	String	ORM : Name of any index constraint to add	
indexed	String	ORM : Whether the join column(s) is indexed	
uniqueKey	String	ORM : Name of any unique constraint to add	
unique	String	ORM : Whether the join column(s) has a unique constraint	
columns	Column[]	ORM : Name of the columns to join our PK to in the join table (when multiple columns used)	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Element(types=org.datanucleus.samples.MyElement.class)
    @Join(table="MYCLASS_ELEMENTS", column="MYCLASS_ELEMENTS_PK")
    Collection myField;
    ...
}
```

@Embedded

Annotation used to define that the field contents is embedded into the same table as this field Maps across to the <embedded> MetaData element. Specified on the **field/method**.

Attribute	Type	Description	Default
ownerMember	String	ORM : The field/property in the embedded object that links back to the owning object (where it has a bidirectional relation)	
nullIndicatorColumn	String	ORM : The column in the embedded object used to judge if the embedded object is null.	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
nullIndicatorValue	String	ORM : The value in the null column to interpret the object as being null.	
members	Persistent[]	ORM : Field/property definitions for this embedding.	

```
@PersistenceCapable
public class MyClass
{
    @Embedded(members={
            @Persistent(name="field1", columns=@Column(name="OTHER_FLD_1")),
            @Persistent(name="field2", columns=@Column(name="OTHER_FLD_2"))
        }
    MyOtherClass myField;
    ...
}

@PersistenceCapable
@EmbeddedOnly
public class MyOtherClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;
}
```

@Columns

Annotation used to define the columns into which a field is persisted. If the field is persisted into a single column then @Column should be used. Specified on the **field/method**.

Attribute	Type	Description	Default
value	Column[]	Array of columns - see @Columns annotation	

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Columns({@Column(name="RED"), @Column(name="GREEN"), @Column(name="BLUE"),
@Column(name="ALPHA")})
    Color myField;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@Column

Annotation used to define that the colum where a field is persisted. Is equivalent to the <column> metadata element when specified under field. Specified on the **field/method** (and within other annotations).

Attribute	Type	Description	Default
name	String	ORM : Name of the column	
target	String	ORM : Column in the other class that this maps to	
targetMember	String	ORM : Field/Property in the other class that this maps to	
jdbcType	String	ORM : JDBC Type to use for persisting into this column	
sqlType	String	ORM : SQL Type to use for persisting into this column	
length	int	ORM : Max length of data to store in this column	
scale	int	ORM : Max number of floating points of data to store in this column	
allowsNull	String	ORM : Whether null is allowed to be persisted into this column	
defaultValue	String	ORM : Default value to persist into this column	
insertValue	String	ORM : Value to insert into this column when it is an "unmapped" column.	
extensions	Extension[]	Vendor extensions	

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Column(name="MYCOL", jdbcType="VARCHAR", length=40)
    String field1;

    ...
}
```

@Index

Annotation used to define that this field is indexed. Is equivalent to the <index> metadata element when specified under field. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	ORM : Name of the index	
unique	String	ORM : Whether the index is unique	

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Index(name="MYFIELD1_IDX")
    String field1;

    @Persistent
    @Index(name="MYFIELD2_IDX", unique="true")
    String field2;

    ...
}
```

@Unique

Annotation used to define that this field has a unique constraint. Is equivalent to the <unique> metadata element when specified under field. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	ORM : Name of the constraint	
deferred	String	ORM : Whether the constraint is deferred	

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Unique(name="MYFIELD1_IDX")
    String field1;

    ...
}
```

@ForeignKey

Annotation used to define the foreign key for a relationship field. Is equivalent to the <foreign-key> metadata element when specified under field. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	ORM : Name of the constraint	
deferred	String	ORM : Whether the constraint is deferred	
unique	String	ORM : Whether the constraint is unique	
deleteAction	ForeignKeyAction	ORM : Action to apply to the FK to be used on deleting	ForeignKeyAction.RESTRICT
updateAction	ForeignKeyAction	ORM : Action to apply to the FK to be used on updating	ForeignKeyAction.RESTRICT

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @ForeignKey(name="MYFIELD1_FK", deleteAction=ForeignKeyAction.RESTRICT)
    String field1;

    ...
}
```

@Extensions

Annotation used to define a set of extensions specific to the JDO2 implementation being used. Specified on the **class** or **field**.

Attribute	Type	Description	Default
value	Extension[]	Array of extensions - see @Extension annotation	

```
@PersistenceCapable
@Extensions({@Extension(vendorName="datanucleus", key="firstExtension",
value="myValue"),
             @Extension(vendorName="datanucleus", key="secondExtension",
value="myValue")})
public class Person
{
    ...
}
```

@Extension

Annotation used to define an extension specific to a particular JDO implementation. Is equivalent to the <extension> metadata element. Specified on the **class** or **field**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
vendorName	String	Name of the JDO vendor	
key	String	Key for the extension	
value	String	Value of the extension	

```
@PersistenceCapable
@Extension(vendorName="DataNucleus", key="RunFast", value="true")
public class Person
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.10.3 JDO MetaData API

JDO Metadata API

When using JDO you need to define which classes are persistent, and also how they are persisted. JDO has allowed XML metadata since its first revision, and introduced support for annotations in JDO 2.1. JDO 2.3 introduces a programmatic API to do the same task.

Defining Metadata for classes

The basic idea behind the Metadata API is that the developer obtains a metadata object from the PersistenceManagerFactory, and adds the definition to that as required, before registering it for use in the persistence process.

```
    PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory(propsFile);
    ...
    JDOMetadata md = pmf.newMetadata();
```

So we have a *JDOMetadata* object and want to define the persistence for our class *mydomain.MyClass*, so we do as follows

```
    PackageMetadata pmd = md.newPackageMetadata("mydomain");
    ClassMetadata cmd = pmd.newClassMetadata("MyClass");
```

So we follow the same structure of the JDO XML Metadata file adding packages to the top level, and classes to the respective package. Note that we could have achieved this by a simple typesafe invocation

```
    ClassMetadata cmd = md.newClassMetadata(MyClass.class);
```

So now we have the class defined, we need to set its key information

```
cmd.setTable("CLIENT").setDetachable(true).setIdentityType(IdentityType.DATASTORE);
    cmd.setPersistenceModifier(ClassPersistenceModifier.PERSISTENCE_CAPABLE);

    InheritanceMetadata inhmd = cmd.newInheritanceMetadata();
    inhmd.setStrategy(InheritanceStrategy.NEW_TABLE);
    DiscriminatorMetadata dmd = inhmd.newDiscriminatorMetadata();
    dmd.setColumn("disc").setValue("Client");
    dmd.setStrategy(DiscriminatorStrategy.VALUE_MAP).setIndexed(Indexed.TRUE);

    VersionMetadata vermd = cmd.newVersionMetadata();
    vermd.setStrategy(VersionStrategy.VERSION_NUMBER);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    vermd.setColumn("version").setIndexed(Indexed.TRUE);
```

And we define also define fields/properties via the API in a similar way

```
    FieldMetadata fmd = cmd.newFieldMetadata("name");
    fmd.setNullValue(NullValue.DEFAULT).setColumn("client_name");
    fmd.setIndexed(true).setUnique(true);
```

Note that, just like with XML metadata, we don't need to add information for all fields since they have their own default persistence settings based on the type of the field.

All that remains is to register the metadata with the persistence process

```
    pmf.registerMetadata(md);
```

Accessing Metadata for classes

Maybe you have a class with its persistence defined in XML or annotations and you want to check its persistence information at runtime. With the JDO Metadata API you can do that

```
ComponentMetadata compmd = pmf.getMetadata("mydomain.MyOtherClass");
```

and we can now inspect the information, casting the *compmd* to either *javax.jdo.metadata.ClassMetadata* or *javax.jdo.metadata.InterfaceMetadata*.

Please note that you cannot currently change metadata retrieved in this way, only view it

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.11 Persistence Unit

Persistence Unit

When designing an application you can usually nicely separate your persistable objects into independent groupings that can be treated separately, perhaps within a different DAO object, if using DAOs. JDO2.1 uses the (JPA1) idea of a *persistence-unit*. A *persistence-unit* provides a convenient way of specifying a set of metadata files, and classes, and jars that contain all classes to be persisted in a grouping. The persistence-unit is named, and the name is used for identifying it. Consequently this name can then be used when defining what classes are to be enhanced, for example.

To define a *persistence-unit* you first need to add a file **persistence.xml** to the *META-INF/* directory of your application jar. This file will be used to define your *persistence-unit*s. Let's show an example

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <!-- Online Store -->
    <persistence-unit name="OnlineStore">
        <class>org.datanucleus.samples.metadata.store.Product</class>
        <class>org.datanucleus.samples.metadata.store.Book</class>
        <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
        <class>org.datanucleus.samples.metadata.store.Customer</class>
        <class>org.datanucleus.samples.metadata.store.Supplier</class>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.h2.Driver"/>
            <property name="datanucleus.ConnectionURL" value="jdbc:h2:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>

    <!-- Accounting -->
    <persistence-unit name="Accounting">
<mapping-file>com/datanucleus/samples/metadata/accounts/package.jdo</mapping-file>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.h2.Driver"/>
            <property name="datanucleus.ConnectionURL" value="jdbc:h2:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>

</persistence>
```

In this example we have defined 2 *persistence-unit*s. The first has the name "OnlineStore" and contains 5 classes (annotated). The second has the name "Accounting" and contains a metadata file called "orm.xml" in a particular package (which will define the classes being part of that unit). This means that once we

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

have defined this we can reference these *persistence-unit*s in our persistence operations.

There are several sub-elements of this *persistence.xml* file

- **provider** - Not used by JDO
- **jar-file** - name of a JAR file to scan for annotated classes to include in this persistence-unit.
- **mapping-file** - name of an XML "mapping" file containing persistence information to be included in this persistence-unit. This is the "JDO" mapping file (**not** the ORM)
- **class** - name of an annotated class to include in this persistence-unit
- **properties** - properties defining the persistence factory to be used.

Use with JDO2.1

JDO2.1 accepts the "persistence-unit" name to be specified at runtime when creating the *PersistenceManagerFactory*, like this

```
Properties props = new Properties();
props.put("datanucleus.PersistenceUnitName", "MyPersistenceUnit");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.1 ORM with JDO

JDO Object/Relational Mapping

When you are using an RDBMS datastore you need to specify how your class will map on to the relational datastore. This part is termed **Object-Relational Mapping**. This is not required for other types of datastore. When you are persisting to RDBMS datastores you are mapping a series of objects into a series of datastore *tables* in a *schema*. These *tables* are interrelated using *foreign-keys*. With JDO2 you can define fully this object-relational mapping in the MetaData (or in annotations if you so wish).

The design of the persistence layer of an application requiring object-relational mapping can be approached in 3 ways.

- Forward Mapping - Here you have a set of model classes, and want to design the datastore schema that will store represent these classes.
- Reverse Mapping - Here you have an existing datastore schema, and want to design your model classes to represent this schema.
- Meet in the Middle Mapping - Here you have a set of model classes and you have an existing datastore schema, and you want to match them up.

DataNucleus can be used in all of these modes, though provides significant assistance for Forward Mapping cases. In particular, when using this mode you can use the DataNucleus SchemaTool to generate the datastore schema, based on a set of input classes and MetaData files. It should be noted though that DataNucleus SchemaTool also provides modes of operation for updating an existing schema, and hence can also be used for Meet in the Middle Mapping. Additionally, it can be used as a validation mechanism when designing your system in Reverse Mapping mode, where it will inform you of inconsistencies between your classes and your datastore schema.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.2 ORM Meta-Data

ORM Meta-Data

JDO defines that MetaData (defined in the MetaData guide) can be found in particular locations in the CLASSPATH, and has a particular format. It also defines that you can split your MetaData for *Object Relational Mapping (ORM)* into separate files if you so wish. So you would define your basic persistence in a file "package.jdo" and then define the MetaData files "package-mysql.orm" (for MySQL), and "package-oracle.orm" (for Oracle). To make use of this JDO 2 Object-Relational Mapping file separation, you must specify the PersistenceManagerFactory property datanucleus.Mapping. If you set this to, for example, *mysql* DataNucleus would look for files such as *package.jdo* and *package-mysql.orm* in the same locations as specified above.

Simple Example

Let us take a sample class and generate MetaData for it. Suppose I have a class as follows

```
package mydomain;

public class Person
{
    /** Title of the Person. */
    String title=null;

    /** Forename of the Person. */
    String forename=null;

    /** Surname of the Person. */
    String surname=null;

    ...
}
```

and I want to use an existing schema. With this case I need to define the table and column names that it maps to. To do this I need to use JDO 2 ORM tags. So I come up with MetaData as follows in package.jdo

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="mydomain">
        <class name="Person" identity-type="datastore">
            <field name="title"/>
            <field name="forename"/>
            <field name="surname"/>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </package>
</jdo>
```

and then I add the ORM information in package-mysql.orm as

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE orm PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Mapping Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_orm_2_0.dtd">
<orm>
    <package name="mydomain">
        <class name="Person" table="PERSON">
            <field name="title">
                <column name="TITLE"/>
            </field>
            <field name="forename">
                <column name="FORENAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="surname">
                <column name="SURNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
        </class>
    </package>
</orm>
```

So you see that our class is being mapped across to a table "PERSON" in the datastore, with columns "TITLE", "FORENAME", "SURNAME". We have also specified that the upper size limit on the forename and surname fields is 100.

Memory utilisation

The XML files are parsed and populated to memory the first time a pesistent operation is executed over a persistent class (e.g. *pm.makePersistent(object)*). If the persistent class has relationships to other persistent classes, the metadata for the classes in the relationships are loaded. In addition to the persistent class and classes in the relationships, all other classes / files that were encountered while searching for the persistent classes are loaded, plus their relationships.

In average, for each persistent class a 3kb of memory is used to hold metadata information. This value will vary according the amount of metadata declared. Although this value can be used as reference in earlier stages of development, you should verify if it corresponds to your persistent classes.

A general formula can be used (with caution) to estimate the amount of memory required:

```
Amount Required = (# of persistent classes) * 3KB
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.3 Schema Mapping

Schema Mapping

You saw in our basic class mapping guide how you define MetaData for a classes basic persistence, notating which fields are persisted. The next step is to define how it maps to the schema of the datastore (in this case RDBMS). The simplest way of mapping is to map each class to its own table. This is the default model in JDO persistence (with the exception of inheritance). If you don't specify the table and column names, then DataNucleus will generate table and column names for you. **You should specify your table and column names if you have an existing schema.** Failure to do so will mean that DataNucleus uses its own names and these will almost certainly not match what you have in the datastore. There are several aspects to cover here

- Table and column names
- Column for datastore identity
- Column(s) for application identity
- Column nullability and default value
- Column Types
- Columns with no field in the class

Tables and Column names

The main thing that developers want to do when they set up the persistence of their data is to control the names of the tables and columns used for storing the classes and fields. This is an essential step when mapping to an existing schema, because it is necessary to map the classes onto the existing database entities. Let's take an example

```
public class Hotel
{
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    ...
}
```

In our case we want to map this class to a table called ESTABLISHMENT, and has columns NAME, DIRECTION, PHONE and NUMBER_OF_ROOMS (amongst other things). So we define our Meta-Data like this

```
<class name="Hotel" table="ESTABLISHMENT">
    <field name="name">
        <column name="NAME"/>
    </field>
    <field name="address">
        <column name="DIRECTION"/>
    </field>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="telephoneNumber">
        <column name="PHONE"/>
    </field>
    <field name="numberOfRooms">
        <column name="NUMBER_OF_ROOMS"/>
    </field>
</class>
```

So we have defined the table and the column names. It should be mentioned that if you don't specify the table and column names then DataNucleus will generate names for the datastore identifiers. The table name will be based on the class name, and the column names will be based on the field names and the role of the field (if part of a relationship).

See also :-

- Identifier Guide - defining the identifiers to use for table/column names
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Column
- Annotations reference for @PrimaryKey

Column names for datastore-identity

When you select *datastore-identity* a surrogate column will be added in the datastore. You need to be able to define the column name if mapping to an existing schema (or wanting to control the schema). So lets say we have the following

```
public class MyClass // persisted to table "MYCLASS"
{
    ...
}

public class MySubClass extends MyClass // persisted to table "MYSUBCLASS"
{
    ...
}
```

We want to define the names of the identity column in "MYCLASS" and "MYSUBCLASS". Here's how we do it

```
<class name="MyClass" table="MYCLASS">
    <datastore-identity>
        <column name="MY_PK_COLUMN"/>
    </datastore-identity>
    ...
</class>
<class name="MySubClass" table="MYSUBCLASS">
    <datastore-identity>
        <column name="MYSUB_PK_COLUMN"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </datastore-identity>
    ...
</class>
```

So we will have a PK column "MY_PK_COLUMN" in the table "MYCLASS", and a PK column "MYSUB_PK_COLUMN" in the table "MYSUBCLASS" (and that corresponds to the "MY_PK_COLUMN" value in "MYCLASS"). We could also do

```
<class name="MyClass" table="MYCLASS">
    <datastore-identity>
        <column name="MY_PK_COLUMN"/>
    </datastore-identity>
    ...
</class>
<class name="MySubClass" table="MYSUBCLASS">
    <inheritance strategy="new-table"/>
    <primary-key>
        <column name="MYSUB_PK_COLUMN"/>
    </primary-key>
    ...
</class>
```

See also :-

- Inheritance Guide - defining how to use inheritance between classes
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Column
- Annotations reference for @PrimaryKey

Column names for application-identity

When you select *application-identity* you have some field(s) that form the "primary-key" of the class. A common situation is that you have inherited classes and each class has its own table, and so the primary-key column names can need defining for each class in the inheritance tree. So lets show an example how to do it

```
public class MyClass // persisted to table "MYCLASS"
{
    long id; // PK field
    ...
}

public class MySubClass extends MyClass // persisted to table "MYSUBCLASS"
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Defining the column name for "MyClass.id" is easy since we use the same as shown previously "column" for the field. Obviously the table "MYSUBCLASS" will also need a PK column. Here's how we define the column mapping

```
<class name="MyClass" identity-type="application" table="MYCLASS">
    <field name="myPrimaryKeyField" primary-key="true">
        <column name="MY_PK_COLUMN"/>
    </field>
    ...
</class>
<class name="MySubClass" identity-type="application" table="MYSUBCLASS">
    <inheritance strategy="new-table"/>
    <primary-key>
        <column name="MYSUB_PK_COLUMN" target="MY_PK_COLUMN"/>
    </primary-key>
    ...
</class>
```

So we will have a PK column "MY_PK_COLUMN" in the table "MYCLASS", and a PK column "MYSUB_PK_COLUMN" in the table "MYSUBCLASS" (and that corresponds to the "MY_PK_COLUMN" value in "MYCLASS"). You can also use

```
<class name="MyClass" identity-type="application" table="MYCLASS">
    <field name="myPrimaryKeyField" primary-key="true">
        <column name="MY_PK_COLUMN"/>
    </field>
    ...
</class>
<class name="MySubClass" identity-type="application" table="MYSUBCLASS">
    <inheritance strategy="new-table">
        <join>
            <column name="MYSUB_PK_COLUMN" target="MY_PK_COLUMN"/>
        </join>
    </inheritance>
    ...
</class>
```

See also :-

- Inheritance Guide - defining how to use inheritance between classes
- MetaData reference for <inheritance> element
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Inheritance
- Annotations reference for @Column
- Annotations reference for @PrimaryKey

Column nullability and default values

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So we've seen how to specify the basic structure of a table, naming the table and its columns, and how to control the types of the columns. We can extend this further to control whether the columns are allowed to contain nulls and to set a default value for a column if we ever have need to insert into it and not specify a particular column. Let's take a related class for our hotel. Here we have a class to model the payments made to the hotel.

```
public class Payment
{
    Customer customer;
    String bankTransferReference;
    String currency;
    double amount;
}
```

In this class we can model payments from a customer of an amount. Where the customer pays by bank transfer we can save the reference number. Since our hotel is in the United Kingdom we want the default currency to be pounds, or to use its ISO4217 currency code "GBP". In addition, since the bank transfer reference is optional we want that column to be nullable. So let's specify the MetaData for the class.

```
<class name="Payment">
    <field name="customer" persistence-capable="persistent" column="CUSTOMER_ID"/>
    <field name="bankTransferReference">
        <column name="TRANSFER_REF" allows-null="true"/>
    </field>
    <field name="currency">
        <column name="CURRENCY" default-value="GBP"/>
    </field>
    <field name="amount" column="AMOUNT"/>
</class>
```

So we make use of the allows-null and default-value attributes. The table, when created by DataNucleus, will then provide the default and nullability that we require.

See also :-

- MetaData reference for <column> element
- Annotations reference for @Column

Column types

DataNucleus will provide a default type for any columns that it creates, but it will allow users to override this default. The default that DataNucleus chooses is always based on the Java type for the field being mapped. For example a Java field of type "int" will be mapped to a column type of INTEGER in RDBMS datastores. Similarly String will be mapped to VARCHAR. To override the default setting (and always the best policy if you are wanting your MetaData to give the same datastore definition with all JDO implementations) you do as follows

```
<class name="Payment">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="customer" persistence-capable="persistent" column="CUSTOMER_ID">
    <field name="bankTransferReference">
        <column name="TRANSFER_REF" jdbc-type="VARCHAR" length="255"
allows-null="true"/>
    </field>
    <field name="currency">
        <column name="CURRENCY" jdbc-type="CHAR" length="3" default-value="GBP"/>
    </field>
    <field name="amount">
        <column name="AMOUNT" jdbc-type="DECIMAL" length="10" scale="2"/>
    </field>
</class>
```

So we have defined TRANSFER_REF to use VARCHAR(255) column type, CURRENCY to use CHAR(3) column type, and AMOUNT to use DECIMAL(10,2) column type. Please be aware that DataNucleus only supports persisting particular Java types to particular JDBC/SQL types. We have demonstrated above the jdbc-type attribute, but there is also an sql-type attribute. This is to be used where you want to map to some specific SQL type (and will not be needed in the vast majority of cases - the jdbc-type should generally be used).

See also :-

- Types Guide - defining persistence of Java types
- RDBMS Types Guide - defining mapping of Java types to available JDBC/SQL types
- MetaData reference for <column> element
- Annotations reference for @Column

Columns with no field in the class

DataNucleus supports mapping of columns in the datastore that have no associated field in the java class. These are useful where you maybe have a table used by other applications and dont use some of the information in your Java model. DataNucleus needs to know about these columns so that it can validate the schema correctly, and also insert particular values when inserting objects into the table. You could handle this by defining your schema yourself so that the particular columns have "DEFAULT" settings, but this way you allow DataNucleus to know about all information. So to give an example

```
<class name="Hotel" table="ESTABLISHMENT">
    <field name="name">
        <column name="NAME"/>
    </field>
    <field name="address">
        <column name="DIRECTION"/>
    </field>
    <field name="telephoneNumber">
        <column name="PHONE"/>
    </field>
    <field name="numberOfRooms">
        <column name="NUMBER_OF_ROOMS"/>
    </field>
    <column name="YEAR_ESTABLISHED" jdbc-type="INTEGER" insert-value="1980"/>
    <column name="MANAGER_NAME" jdbc-type="VARCHAR" insert-value="N/A"/>
</class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So in this example our table "ESTABLISHMENT" has the columns associated with the specified fields and also has columns "YEAR_ESTABLISHED" (that is INTEGER-based and will be given a value of "1980" on any inserts) and "MANAGER_NAME" (VARCHAR-based and will be given a value of "N/A" on any inserts).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.4 Datastore Identifiers

JDO Datastore Identifiers

A datastore identifier is a simple name of a database object, such as a column, table, index, or view, and is composed of a sequence of letters, digits, and underscores (_) that represents it's name. DataNucleus allows users to specify the names of tables, columns, indexes etc but if the user doesn't specify these DataNucleus will generate names. Generation of identifier names is controlled by an IdentifierFactory, and DataNucleus provides a default implementation. You can provide your own IdentifierFactory plugin to give your own preferred naming if so desired. You set the *IdentifierFactory* by setting the PMF property *datanucleus.identifierFactory*. Set it to the symbolic name of the factory you want to use. JDO doesnt define what the names of datastore identifiers should be but DataNucleus provides 3 factories for your use.

- jpox IdentifierFactory (default for JDO persistence)
- jpox2 IdentifierFactory
- jpa IdentifierFactory (default for JPA persistence)

In describing the different possible naming conventions available out of the box with DataNucleus we'll use the following example

```
class MyClass
{
    String myField1;
    Collection<MyElement> elements1; // Using join table
    Collection<MyElement> elements2; // Using foreign-key
}

class MyElement
{
    String myElementField;
    MyClass myClass2;
}
```

IdentifierFactory 'jpox'

The default *IdentifierFactory* (when using JDO persistence) goes by the name "jpox" and provides a reasonable default naming of datastore identifiers using the class and field names as its basis. DataNucleus has used this naming convention for all versions.

Using the example above, the rules in this *IdentifierFactory* mean that, assuming that the user doesnt specify any <column> elements :-

- *MyClass* will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MY_FIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- *MyClass.elements1* will be persisted into a join table called **MYCLASS_ELEMENTS1**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_ID_OID** (FK to owner table) and **MYELEMENT_ID_EID** (FK to element table)
- MYCLASS_ELEMENTS1 will have column names like **STRING_ELE**, **STRING_KEY**, **STRING_VAL** for non-PC elements/keys/values of collections/maps
- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID_OID** (FK to owner) table
- Any discriminator column will be called **DISCRIMINATOR**
- Any index column in a List will be called **INTEGER_IDX**
- Any adapter column added to a join table to form part of the primary key will be called **ADPT_PK_IDX**
- Any version column for a table will be called **OPT_VERSION**

IdentifierFactory 'jpox2'

The *IdentifierFactory* "jpox2" changes a few things over the default "jpox" factory, attempting to make the naming more concise and consistent (we retain "jpox" for backwards compatibility).

Using the same example above, the rules in this *IdentifierFactory* mean that, assuming that the user doesnt specify any <column> elements :-

- *MyClass* will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MYFIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**
- *MyClass.elements1* will be persisted into a join table called **MYCLASS_ELEMENTS1**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_ID_OID** (FK to owner table) and **MYELEMENT_ID_EID** (FK to element table)
- MYCLASS_ELEMENTS1 will have column names like **STRING_ELE**, **STRING_KEY**, **STRING_VAL** for non-PC elements/keys/values of collections/maps
- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID_OID** (FK to owner) table
- Any discriminator column will be called **DISCRIMINATOR**
- Any index column in a List will be called **IDX**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

IdentifierFactory 'jpoxCompatibility'

This *IdentifierFactory* exists for backward compatibility with JPOX 1.2.0. If you experience changes of

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

schema identifiers when migrating from JPOX 1.2.0 to datanucleus, you should give this one a try.

Schema compatibility between JPOX 1.2.0 and datanucleus had been broken e.g. by the number of characters used in hash codes when truncating identifiers: this has changed from 2 to 4.

IdentifierFactory 'jpa'

The *IdentifierFactory* "jpa" aims at providing a naming policy consistent with the "JPA1" specification.

Using the same example above, the rules in this *IdentifierFactory* mean that, assuming that the user doesnt specify any <column> elements :-

- *MyClass* will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MYFIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**
- *MyClass.elements1* will be persisted into a join table called **MYCLASS_MYELEMENT**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_MYCLASS_ID** (FK to owner table) and **ELEMENTS1_ELEMENT_ID** (FK to element table)
- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID** (FK to owner) table
- Any discriminator column will be called **DTYPE**
- Any index column in a List for field *MyClass.myField1* will be called **MYFIELD1_ORDER**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

IdentifierFactory - Controlling the Case

The underlying datastore will define what case of identifiers are accepted. By default, DataNucleus will capitalise names (assuming that the datastore supports it). You can however influence the case used for identifiers. This is specifiable with the PMF property *datanucleus.identifier.case*, having the following values

- UpperCase: identifiers are in upper case
- LowerCase: identifiers are in lower case
- PreserveCase: No case changes are made to the name of the identifier provided by the user (class name or jdo metadata).

Please be aware that some datastores only support UPPERCASE or lowercase identifiers and so setting this parameter may have no effect if your database doesn't support that option.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.5 Secondary Tables

JDO Secondary Tables

The standard JDO persistence strategy is to persist an object of a class into its own table. In some situations you may wish to map the class to a primary table as well as one or more secondary tables. For example when you have a Java class that could have been split up into 2 separate classes yet, for whatever reason, has been written as a single class, however you have a legacy datastore and you need to map objects of this class into 2 tables. JDO allows persistence of fields of a class into secondary tables.

The process for managing this situation is best demonstrated with an example. Let's suppose we have a class that represents a Printer. The Printer class contains within it various attributes of the toner cartridge. So we have

```
package com.mydomain.samples.secondarytable;

public class Printer
{
    long id;
    String make;
    String model;

    String tonerModel;
    int tonerLifetime;

    /**
     * Constructor.
     * @param make Make of printer (e.g Hewlett-Packard)
     * @param model Model of Printer (e.g LaserJet 1200L)
     * @param tonerModel Model of toner cartridge
     * @param tonerLifetime lifetime of toner (number of prints)
     */
    public Printer(String make, String model, String tonerModel, int tonerLifetime)
    {
        this.make = make;
        this.model = model;
        this.tonerModel = tonerModel;
        this.tonerLifetime = tonerLifetime;
    }

}
```

Now we have a database schema that has 2 tables (PRINTER and PRINTER_TONER) in which to store objects of this class. So we need to tell DataNucleus to perform this mapping. So we define the MetaData for the Printer class like this

```
<class name="Printer" table="PRINTER">
    <join table="PRINTER_TONER" column="PRINTER_REFID"/>

    <field name="id" primary-key="true">
```

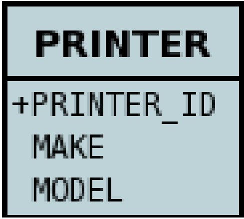
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <column name="PRINTER_ID"/>
    </field>
    <field name="make">
        <column name="MAKE"/>
    </field>
    <field name="model">
        <column name="MODEL"/>
    </field>
    <field name="tonerModel" table="PRINTER_TONER">
        <column name="MODEL"/>
    </field>
    <field name="tonerLifetime" table="PRINTER_TONER">
        <column name="LIFETIME"/>
    </field>
</class>
```

So here we have defined that objects of the Printer class will be stored in the primary table PRINTER. In addition we have defined that some fields are stored in the table PRINTER_TONER. This is achieved by way of

- We will store tonerModel and tonerLifetime in the table PRINTER_TONER. This is achieved by using <field table="PRINTER_TONER">
- The table PRINTER_TONER will use a primary key column called PRINTER_REFID. This is achieved by using <join table="PRINTER_TONER" column="PRINTER_REFID"/>

This results in the following database tables :-



So we now have our primary and secondary database tables. The primary key of the PRINTER_TONER table serves as a foreign key to the primary class. Whenever we persist a Printer object a row will be inserted into both of these tables.

Specifying the primary key

You saw above how we defined the column name that will be the primary key of the secondary table (the PRINTER_REFID column). What we didn't show is how to specify the name of the primary key constraint to be generated. To do this you change the MetaData to

```
<class name="Printer" identity-type="datastore" table="PRINTER">
    <join table="PRINTER_TONER" column="PRINTER_REFID">
        <primary-key name="TONER_PK"/>
    </join>

    <field name="id" primary-key="true">
        <column name="PRINTER_ID"/>
    </field>
    <field name="make">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <column name="MAKE"/>
    </field>
    <field name="model">
        <column name="MODEL"/>
    </field>
    <field name="tonerModel" table="PRINTER_TONER">
        <column name="MODEL"/>
    </field>
    <field name="tonerLifetime" table="PRINTER_TONER">
        <column name="LIFETIME"/>
    </field>
</class>
```

So this will create the primary key constraint with the name "TONER_PK".

See also :-

- MetaData reference for <primary-key> element
- MetaData reference for <join> element
- Annotations reference for @PrimaryKey
- Annotations reference for @Join

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.6 Embedded Objects

JDO Embedded Objects

The JDO persistence strategy typically involves persisting the fields of any class into its own table, and representing any relationships from the fields of that class across to other tables. There are occasions when this is undesirable, maybe due to an existing datastore schema, or because a more convenient datastore model is required. JDO allows the persistence of fields as embedded typically into the same table as the "owning" class.

One important decision when defining objects of a type to be embedded into another type is whether objects of that type will ever be persisted in their own right into their own table, and have an identity. JDO2 provides a MetaData attribute that you can use to signal this.

```
<jdo>
    <package name="com.mydomain.samples.embedded">
        <class name="MyClass" embedded-only="true">
            ...
        </class>

    </package>
</jdo>
```

With the above MetaData (using the embedded-only attribute), in our application any objects of the class MyClass cannot be persisted in their own right. They can only be embedded into other objects.

JDO2's definition of embedding encompasses several types of fields. These are described below

- Embedded PersistenceCapable objects - where you have a 1-1 relationship and you want to embed the other PersistenceCapable into the same table as the your object.
- Embedded Nested PersistenceCapable objects - like the first example except that the otehr object also has another PersistenceCapable object that also should be embedded
- Embedded Collection elements - where you want to embed the elements of a collection into a join table (instead of persisting them into their own table)
- Embedded Map keys/values - where you want to embed the keys/values of a map into a join table (instead of persisting them into their own table)

Embedding PersistenceCapable objects

In a typical 1-1 relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and a foreign key is managed between them. With JDO2 and DataNucleus you can persist the related PersistenceCapable object as embedded into the same table. This results in a single table in the datastore rather than one for each of the 2 classes.

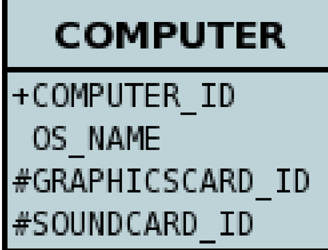
Let's take an example. We are modelling a Computer, and in our simple model our Computer has a graphics card and a sound card. So we model these cards using a ComputerCard class. So our classes

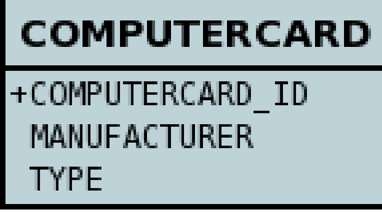
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

become

```
public class Computer
{
    private String operatingSystem;

    private ComputerCard graphicsCard;

    private ComputerCard soundCard;

    public Computer(String osName,
                    ComputerCard graphics,
                    ComputerCard sound)
    {
        this.operatingSystem = osName;
        this.graphicsCard = graphics;
        this.soundCard = sound;
    }

    ...
}

public class ComputerCard
{
    public static final int ISA_CARD = 0;
    public static final int PCI_CARD = 1;
    public static final int AGP_CARD = 2;

    private String manufacturer;

    private int type;

    public ComputerCard(String manufacturer,
                        int type)
    {
        this.manufacturer = manufacturer;
        this.type = type;
    }

    ...
}
```

The traditional (default) way of persisting these classes would be to have a table to represent each class. So our datastore will look like this





However we decide that we want to persist Computer objects into a table called COMPUTER and we also want to persist the PC cards into the same table. We define our MetaData like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="com.mydomain.samples.embedded">
        <class name="Computer" identity-type="datastore" table="COMPUTER">
            <field name="operatingSystem">
                <column name="OS_NAME" length="40" jdbc-type="CHAR"/>
            </field>
            <field name="graphicsCard" persistence-modifier="persistent">
                <embedded null-indicator-column="GRAPHICS_MANUFACTURER">
                    <field name="manufacturer" column="GRAPHICS_MANUFACTURER"/>
                    <field name="type" column="GRAPHICS_TYPE"/>
                </embedded>
            </field>
            <field name="soundCard" persistence-modifier="persistent">
                <embedded null-indicator-column="SOUND_MANUFACTURER">
                    <field name="manufacturer" column="SOUND_MANUFACTURER"/>
                    <field name="type" column="SOUND_TYPE"/>
                </embedded>
            </field>
        </class>

        <class name="ComputerCard" table="COMPUTER_CARD">
            <field name="manufacturer"/>
            <field name="type"/>
        </class>
    </package>
</jdo>
```

So here we will end up with a TABLE called "COMPUTER" with columns "COMPUTER_ID", "OS_NAME", "GRAPHICS_MANUFACTURER", "GRAPHICS_TYPE", "SOUND_MANUFACTURER", "SOUND_TYPE". If we call makePersistent() on any objects of type Computer, they will be persisted into this table.

COMPUTER
+COMPUTER_ID
OS_NAME
GRAPHICS_MANUFACTURER
GRAPHICS_TYPE
SOUND_MANUFACTURER
SOUND_TYPE

You will notice in the MetaData our use of the attribute null-indicator-column. This is used when retrieving objects from the datastore and detecting if it is a NULL embedded object. In the case we have here, if the column GRAPHICS_MANUFACTURER is null at retrieval, then the embedded "graphicsCard" field will be set as null. Similarly for the "soundCard" field when SOUND_MANUFACTURER is null.

If the ComputerCard class above has a reference back to the related Computer, JDO2 defines a mechanism whereby this will be populated. You would add the XML element owner-field to the <embedded> tag defining the field within ComputerCard that represents the Computer it relates to. When this is specified DataNucleus will populate it automatically, so that when you retrieve the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Computer and access the ComputerCard objects within it, they will have the link in place.

It should be noted that in this latter (embedded) case we can still persist objects of type ComputerCard into their own table - the MetaData definition for ComputerCard is used for the table definition in this case.

Please note that if, instead of specifying the <embedded> block we had specified **embedded** in the field element we would have ended up with the same thing, just that the fields and columns would have been mapped using their default mappings, and that the <embedded> provides control over how they are mapped.

DataNucleus supports embedded PC objects with the following proviso :-

- Embedded PC objects cannot have inheritance (this restriction will hopefully be removed in the future, allowing a discriminator).

See also :-

- MetaData reference for <embedded> element
- Annotations reference for @Embedded

Embedding Nested PersistenceCapable objects

In the above example we had an embedded PersistenceCapable object within a persisted object. What if our embedded PersistenceCapable object also contain another PersistenceCapable object. So, using the above example what if ComputerCard contains an object of type Connector ?

```
public class ComputerCard
{
    ...

    Connector connector;

    public ComputerCard(String manufacturer,
                        int type,
                        Connector conn)
    {
        this.manufacturer = manufacturer;
        this.type = type;
        this.connector = conn;
    }

    ...
}

public class Connector
{
    int type;
}
```

Well we want to store all of these objects into the same record in the COMPUTER table.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="com.mydomain.samples.embedded">
        <class name="Computer" identity-type="datastore" table="COMPUTER">
            <field name="operatingSystem">
                <column name="OS_NAME" length="40" jdbc-type="CHAR"/>
            </field>
            <field name="graphicsCard" persistence-modifier="persistent">
                <embedded null-indicator-column="GRAPHICS_MANUFACTURER">
                    <field name="manufacturer" column="GRAPHICS_MANUFACTURER"/>
                    <field name="type" column="GRAPHICS_TYPE"/>
                    <field name="connector">
                        <embedded>
                            <field name="type" column="GRAPHICS_CONNECTOR_TYPE"/>
                        </embedded>
                    </field>
                </embedded>
            </field>
            <field name="soundCard" persistence-modifier="persistent">
                <embedded null-indicator-column="SOUND_MANUFACTURER">
                    <field name="manufacturer" column="SOUND_MANUFACTURER"/>
                    <field name="type" column="SOUND_TYPE"/>
                    <field name="connector">
                        <embedded>
                            <field name="type" column="SOUND_CONNECTOR_TYPE"/>
                        </embedded>
                    </field>
                </embedded>
            </field>
        </class>

        <class name="ComputerCard" table="COMPUTER_CARD">
            <field name="manufacturer"/>
            <field name="type"/>
        </class>

        <class name="Connector" embedded-only="true">
            <field name="type"/>
        </class>
    </package>
</jdo>
```

So we simply nest the embedded definition of the Connector objects within the embedded definition of the ComputerCard definitions for Computer. JDO2 supports this to as many levels as you require! The Connector objects will be persisted into the GRAPHICS_CONNECTOR_TYPE, and SOUND_CONNECTOR_TYPE columns in the COMPUTER table.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

COMPUTER
+COMPUTER_ID
OS_NAME
GRAPHICS_MANUFACTURER
GRAPHICS_TYPE
GRAPHICS_CONNECTOR_TYPE
SOUND_MANUFACTURER
SOUND_TYPE
SOUND_CONNECTOR_TYPE

Embedding Collection Elements

In a typical 1-N relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and either a join table or a foreign key is used to relate them. With JDO2 and DataNucleus you have a variation on the join table relation where you can persist the objects of the "N" side into the join table itself so that they don't have their own identity, and aren't stored in the table for that class. This is supported in DataNucleus with the following provisos

- Embedded elements cannot have inheritance (this may be allowed in the future)
- When retrieving embedded elements, all fields are retrieved in one call. That is, fetch plans are not utilised. This is because the embedded element has no identity so we have to retrieve all initially.

It should be noted that where the collection "element" is not *PersistenceCapable* or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to *PersistenceCapable* elements only. DataNucleus doesn't support the embedding of reference type objects currently.

Let's take an example. We are modelling a Network, and in our simple model our Network has collection of Devices. So we define our classes as

```
public class Network
{
    private String name;

    private Collection devices = new HashSet();

    public Network(String name)
    {
        this.name = name;
    }

    ...
}

public class Device
{
    private String name;

    private String ipAddress;

    public Device(String name,
                  String addr)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

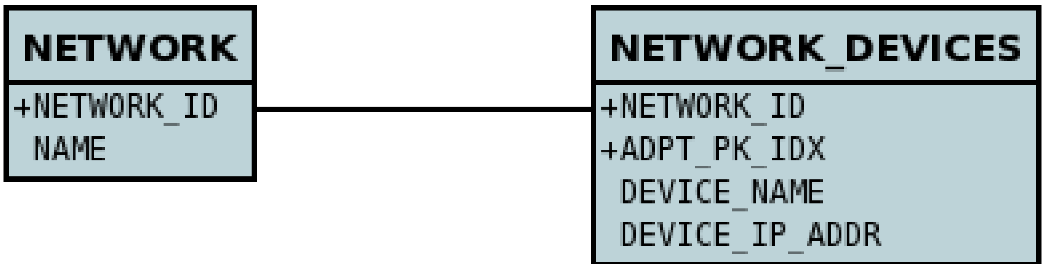
```
    {
        this.name = name;
        this.ipAddress = addr;
    }

    ...
}
```

We decide that instead of Device having its own table, we want to persist them into the join table of its relationship with the Network since they are only used by the network itself. We define our MetaData like this

```
<jdo>
    <package name="com.mydomain.samples.embedded">
        <class name="Network" identity-type="datastore" table="NETWORK">
            <field name="name">
                <column name="NAME" length="40" jdbc-type="VARCHAR"/>
            </field>
            <field name="devices" persistence-modifier="persistent"
table="NETWORK_DEVICES">
                <collection element-type="com.mydomain.samples.embedded.Device"/>
                <join>
                    <column name="NETWORK_ID"/>
                </join>
                <element>
                    <embedded>
                        <field name="name">
                            <column name="DEVICE_NAME" allows-null="true"/>
                        </field>
                        <field name="ipAddress">
                            <column name="DEVICE_IP_ADDR" allows-null="true"/>
                        </field>
                    </embedded>
                </element>
            </field>
        </class>

        <class name="Device" table="DEVICE" embedded-only="true">
            <field name="name">
                <column name="NAME"/>
            </field>
            <field name="ipAddress">
                <column name="IP_ADDRESS"/>
            </field>
        </class>
    </package>
</jdo>
```

So here we will end up with a table called "NETWORK" with columns "NETWORK_ID", and "NAME", and a table called "NETWORK_DEVICES" with columns "NETWORK_ID", "ADPT_PK_IDX", "DEVICE_NAME", "DEVICE_IP_ADDR". When we persist a Network object, any devices are persisted into the NETWORK_DEVICES table.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Please note that if, instead of specifying the <embedded> block we had specified **embedded-element** in the collection element we would have ended up with the same thing, just that the fields and columns would be mapped using their default mappings, and that the <embedded> provides control over how they are mapped.

You note that in our example above DataNucleus has added an extra column "ADPT_PK_IDX" to provide the primary key of the join table now that we're storing the elements as embedded. A variation on this would have been if we wanted to maybe use the "DEVICE_IP_ADDR" as the other part of the primary key, in which case the "ADPT_PK_IDX" would not be needed. You would specify this as follows

```
            <field name="devices" persistence-modifier="persistent"
table="NETWORK_DEVICES">
                <collection element-type="com.mydomain.samples.embedded.Device"/>
                <join>
                    <primary-key name="NETWORK_DEV_PK">
                        <column name="NETWORK_ID"/>
                        <column name="DEVICE_IP_ADDR"/>
                    </primary-key>
                    <column name="NETWORK_ID"/>
                </join>
                <element>
                    <embedded>
                        <field name="name">
                            <column name="DEVICE_NAME" allows-null="true"/>
                        </field>
                        <field name="ipAddress">
                            <column name="DEVICE_IP_ADDR" allows-null="true"/>
                        </field>
                    </embedded>
                </element>
            </field>
```

This results in the join table only having the columns "NETWORK_ID", "DEVICE_IP_ADDR", and "DEVICE_NAME", and having a primary key as the composite of "NETWORK_ID" and "DEVICE_IP_ADDR".

See also :-

- MetaData reference for <embedded> element
- MetaData reference for <element> element
- MetaData reference for <join> element
- Annotations reference for @Embedded

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- Annotations reference for @Element

Embedding Map Keys/Values

In a typical 1-N map relationship between classes, the classes in the relationship are persisted to their own table, and a join table forms the map linkage. With JDO2 amd DataNucleus you have a variation on the join table relation where you can persist either the key class or the value class, or both key class and value class into the join table. This is supported in DataNucleus with the following provisos

- Embedded keys/values cannot have inheritance (this may be allowed in the future)
- When retrieving embedded keys/values, all fields are retrieved in one call. That is, fetch plans are not utilised. This is because the embedded key/value has no identity so we have to retrieve all initially.

It should be noted that where the map "key"/"value" is not *PersistenceCapable* or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to *PersistenceCapable* keys/values only. DataNucleus doesn't support embedding reference type elements currently.

Let's take an example. We are modelling a FilmLibrary, and in our simple model our FilmLibrary has map of Films, keyed by a String alias. So we define our classes as

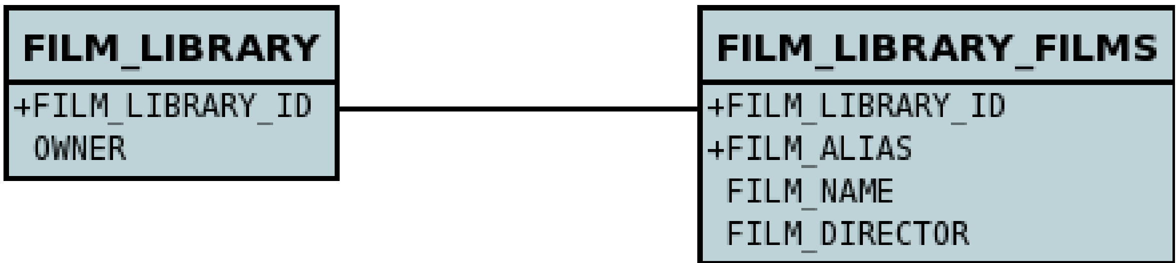
```
public class FilmLibrary
{
    private String owner;

    private Map films = new HashMap();

    public FilmLibrary(String owner)
    {
        this.owner = owner;
    }

    ...
}

public class Film
{
    private String name;

    private String director;

    public Film(String name, String director)
    {
        this.name = name;
        this.director = director;
    }

    ...
}
```

We decide that instead of Film having its own table, we want to persist them into the join table of its map relationship with the FilmLibrary since they are only used by the library itself. We define our MetaData like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="com.mydomain.samples.embedded">
        <class name="FilmLibrary" identity-type="datastore" table="FILM_LIBRARY">
            <field name="owner">
                <column name="OWNER" length="40" jdbc-type="VARCHAR"/>
            </field>
            <field name="films" persistence-modifier="persistent"
table="FILM_LIBRARY_FILMS">
                <map key-type="java.lang.String"
value-type="com.mydomain.samples.embedded.Film"/>
                <join>
                    <column name="FILM_LIBRARY_ID"/>
                </join>
                <key>
                    <column name="FILM_ALIAS"/>
                </key>
                <value>
                    <embedded>
                        <field name="name">
                            <column name="FILM_NAME"/>
                        </field>
                        <field name="director">
                            <column name="FILM_DIRECTOR" allows-null="true"/>
                        </field>
                    </embedded>
                </value>
            </field>
        </class>

        <class name="Film" embedded-only="true">
            <field name="name"/>
            <field name="director"/>
        </class>
    </package>
</jdo>
```

So here we will end up with a table called "FILM_LIBRARY" with columns "FILM_LIBRARY_ID", and "OWNER", and a table called "FILM_LIBRARY_FILMS" with columns "FILM_LIBRARY_ID", "FILM_ALIAS", "FILM_NAME", "FILM_DIRECTOR". When we persist a FilmLibrary object, any films are persisted into the FILM_LIBRARY_FILMS table.



Please note that if, instead of specifying the <embedded> block we had specified **embedded-key** of **embedded-value** in the map element we would have ended up with the same thing, just that the fields and columns would be mapped using their default mappings, and that the <embedded> provides control over how they are mapped.

See also :-

- MetaData reference for <embedded> element
- MetaData reference for <key> element
- MetaData reference for <value> element
- MetaData reference for <join> element
- Annotations reference for @Embedded
- Annotations reference for @Key
- Annotations reference for @Value

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.7 Serialised Objects

JDO Serialising Objects

JDO2 provides a way for users to specify that a field will be persisted *serialised*. This is of use, for example, to collections/maps/arrays which typically are stored using join tables or foreign-keys to other records. By specifying that a field is serialised a column will be added to store that field and the field will be serialised into it.

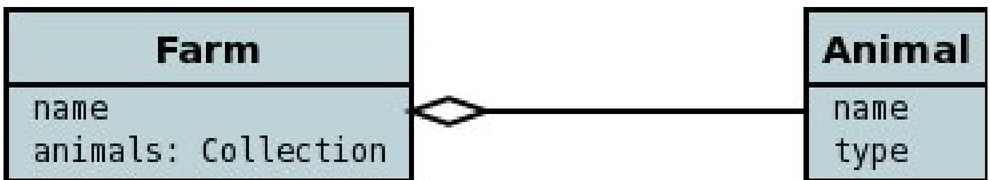
JDO2's definition of serialising encompasses several types of fields. These are described below

- Serialised Array fields - where you want to serialise the array into a single BLOB column.
- Serialised Collection fields - where you want to serialise the collection into a single BLOB column.
- Serialised Collection elements - where you want to serialise the collection elements into a single column in a join table.
- Serialised Map fields - where you want to serialise the map into a single BLOB column
- Serialised Map keys/values - where you want to serialise the map keys and/or values into single column(s) in a join table.
- Serialised PersistenceCapable fields - where you want to serialise a PC object into a single BLOB column.
- Serialised Reference (Interface/Object) fields - where you want to serialise a reference field into a single BLOB column.

Perhaps the most important thing to bear in mind when deciding to serialise a field is that that object must implement *java.io.Serializable*.

Serialised Collections

Collections are usually persisted by way of either a *join table*, or by use of a *foreign-key* in the element table. In some situations it is required to store the whole collection in a single column in the table of the class being persisted. This prohibits the querying of such a collection, but will persist the collection in a single statement. Let's take an example. We have the following classes



and we want the *animals* collection to be serialised into a single column in the table storing the Farm class, so we define our MetaData like this

```
<class name="Farm" table="FARM">
    <datastore-identity column="ID"/>
    <field name="name" column="NAME"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="animals" serialized="true">
        <collection element-type="Animal"/>
        <column name="ANIMALS"/>
    </field>
</class>
<class name="Animal">
    <field name="name"/>
    <field name="type"/>
</class>
```

So we make use of the *serialized* attribute of <field>. This specification results in a table like this

Provisos to bear in mind are

- Queries cannot be performed on collections stored as serialised.

There are some other combinations of MetaData tags that result in serialising of the whole collection in the same way. These are as follows

- **Collection of non-*PersistenceCapable* elements, and no <join> is specified.** Since the elements don't have a table of their own, the only option is to serialise the whole collection and it appears as a single BLOB field in the table of the main class.
- **Collection of *PersistenceCapable* elements, with "embedded-element" set to true and no <join> is specified.** Since the elements are embedded and there is no join table, then the whole collection is serialised as above.

See also :-

- MetaData reference for <field> element
- Annotations reference for @Persistent
- Annotations reference for @Serialized

Serialised Collection Elements

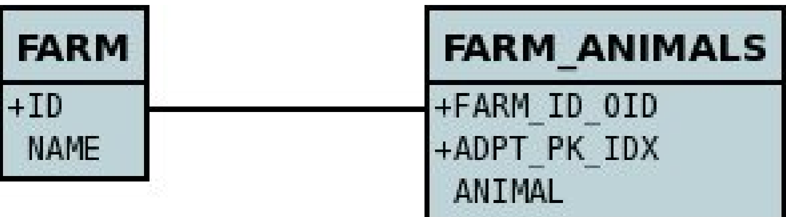
Collections are usually persisted by way of either a *join table*, or by use of a *foreign-key* in the element table. In some situations you may want to serialise the element into a single column in the join table. Let's take an example. We have the same classes as in the previous case and we want the *animals* collection to be stored in a join table, and the element serialised into a single column storing the "Animal" object. We define our MetaData like this

```
<class name="Farm" table="FARM">
    <datastore-identity column="ID"/>
    <field name="name">
        <column name="NAME"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </field>
    <field name="animals" table="FARM_ANIMALS">
        <collection element-type="Animal" serialised-element="true"/>
        <join column="FARM_ID_OID"/>
    </field>
</class>
<class name="Animal">
    <field name="name"/>
    <field name="type"/>
</class>
```

So we make use of the *serialized-element* attribute of <collection>. This specification results in tables like this



Provisos to bear in mind are

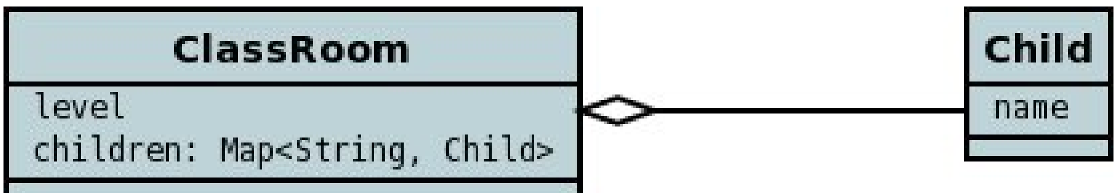
- Queries cannot be performed on collection elements stored as serialised.

See also :-

- MetaData reference for <collection> element
- MetaData reference for <join> element
- Annotations reference for @Element

Serialised Maps

Maps are usually persisted by way of a *join table*, or very occasionaly using a *foreign-key* in the value table. In some situations it is required to store the whole map in a single column in the table of the class being persisted. This prohibits the querying of such a map, but will persist the map in a single statement. Let's take an example. We have the following classes



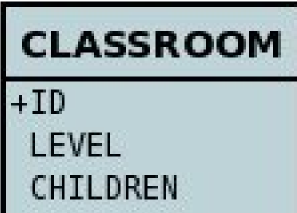
and we want the *children* map to be serialised into a single column in the table storing the ClassRoom class, so we define our MetaData like this

```
<class name="ClassRoom">
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="children" serialized="true">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <map key-type="java.lang.String" value-type="Child"/>
        <column name="CHILDREN"/>
    </field>
</class>
<class name="Child"/>
```

So we make use of the *serialized* attribute of <field>. This specification results in a table like this



Provisos to bear in mind are

- Queries cannot be performed on maps stored as serialised.

There are some other combinations of MetaData tags that result in serialising of the whole map in the same way. These are as follows

- **Map<non-*PersistenceCapable*, non-*PersistenceCapable*>, and no <join> is specified.** Since the keys/values don't have a table of their own, the only option is to serialise the whole map and it appears as a single BLOB field in the table of the main class.
- **Map<non-*PersistenceCapable*, *PersistenceCapable*>, with "embedded-value" set to true and no <join> is specified.** Since the keys/values are embedded and there is no join table, then the whole map is serialised as above.

See also :-

- MetaData reference for <map> element
- Annotations reference for @Key
- Annotations reference for @Value
- Annotations reference for @Serialized

Serialised Map Keys/Values

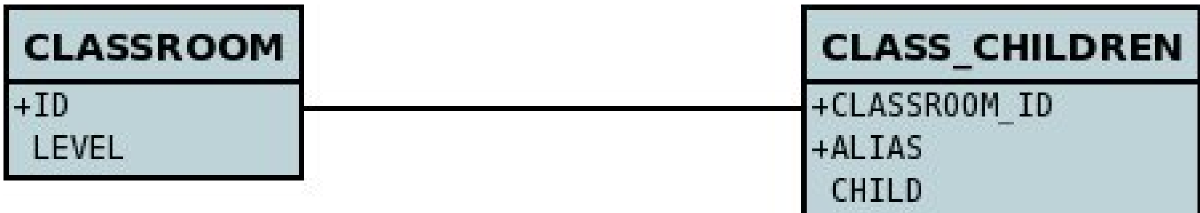
Maps are usually persisted by way of a *join table*, or very occasionaly using a *foreign-key* in the value table. In the join table case you have the option of serialising the keys and/or the values each into a single (BLOB) column in the join table. This is performed in a similar way to serialised elements for collections, but this time using the "serialized-key", "serialized-value" attributes. We take the example in the previous section, with "a classroom of children" and the children stored in a map field. This time we want to serialise the child object into the join table of the map

```
<class name="ClassRoom">
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="children" table="CLASS_CHILDREN">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <map key-type="java.lang.String" value-type="Child"
serialized-value="true"/>
        <join column="CLASSROOM_ID"/>
        <key column="ALIAS"/>
        <value column="CHILD"/>
    </field>
</class>
<class name="Child"/>
```

So we make use of the *serialized-value* attribute of <map>. This results in a schema like this



Provisos to bear in mind are

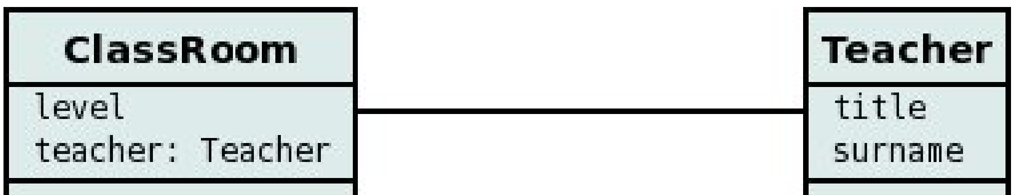
- Queries cannot be performed on map keys/values stored as serialised.

See also :-

- MetaData reference for <map> element
- MetaData reference for <join> element
- MetaData reference for <key> element
- MetaData reference for <value> element
- Annotations reference for @Key
- Annotations reference for @Value

Serialised PersistenceCapable Fields

A field that is a PersistenceCapable object is typically stored as a foreign-key relation between the container object and the contained object. In some situations it is not necessary that the contained object has an identity of its own, and for efficiency of access the contained object is required to be stored in a BLOB column in the containing object's datastore table. Let's take an example. We have the following classes



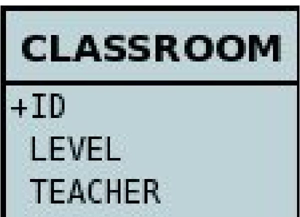
and we want the *teacher* object to be serialised into a single column in the table storing the ClassRoom class, so we define our MetaData like this

```
<class name="ClassRoom">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="teacher" serialized="true">
        <column name="TEACHER"/>
    </field>
</class>
```

So we make use of the *serialized* attribute of <field>. This specification results in a table like this

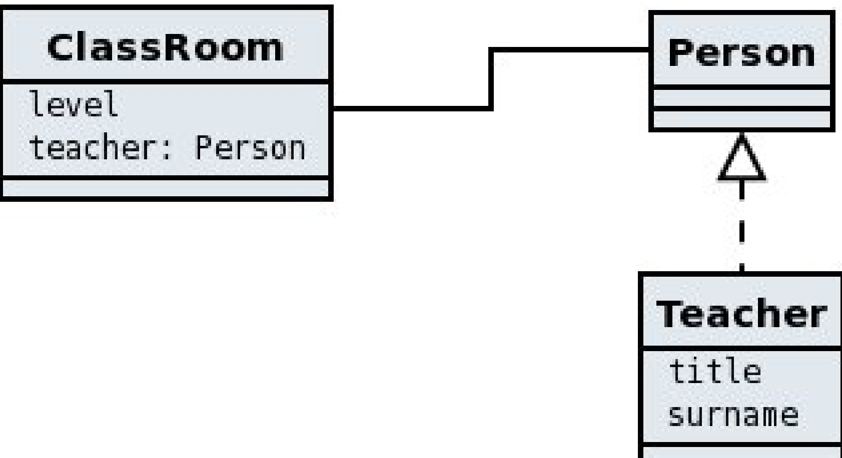


Provisos to bear in mind are

- Queries cannot be performed on PersistenceCapable objects stored as serialised.

Serialised Reference (Interface/Object) Fields

A reference (Interface/Object) field is typically stored as a foreign-key relation between the container object and the contained implementation of the reference. In some situations it is not necessary that the contained object has an identity of its own, and for efficiency of access the contained object is required to be stored in a BLOB column in the containing object's datastore table. Let's take an example using an interface field. We have the following classes



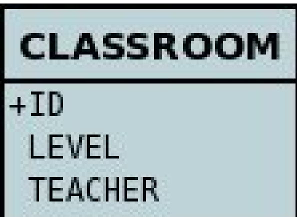
and we want the *teacher* object to be serialised into a single column in the table storing the ClassRoom class, so we define our MetaData like this

```
<class name="ClassRoom">
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="teacher" serialized="true">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <column name="TEACHER"/>
    </field>
</class>
<class name="Teacher">
</class>
```

So we make use of the *serialized* attribute of <field>. This specification results in a table like this



Provisos to bear in mind are

- Queries cannot be performed on Reference (Interface/Object) fields stored as serialised.

See also :-

- MetaData reference for <implements> element
- Annotations reference for @Serialized

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.8 Constraints

Constraints

A datastore often provides ways of constraining the storage of data to maintain relationships and improve performance. These are known as *constraints* and they come in various forms. These are :-

- Indexes - these are used to mark fields that are referenced often as indexes so that when they are used the performance is optimised.
- Unique constraints - these are placed on fields that should have a unique value. That is only one object will have a particular value.
- Foreign-Keys - these are used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.
- Primary-Keys - allow the PK to be set, and also to have a name.

Indexes

The majority of datastores provide the ability to have indexes defined to give performance benefits. With RDBMS the indexes are specified on the table and the indexes to the rows are stored separately. In the same way an ODBMS typically allows indexes to be specified on the fields of the class, and these are managed by the datastore. JDO 2 provides a mechanism for defining indexes, and hence if a developer knows that a particular field is going to be highly used for querying, they can select that field to be indexed in their (JDO) persistence solution. Let's take an example class, and show how to specify this

```
public class Booking
{
    private int bookingType;
    ...
}
```

We decide that our bookingType is going to be highly used and we want to index this in the persistence tool. To do this we define the Meta-Data for our class as

```
<class name="Booking">
    <field name="bookingType">
        <index name="BOOKING_TYPE_INDEX"/>
    </field>
</class>
```

This will mean that DataNucleus will create an index in the datastore for the field and the index will have the name BOOKING_TYPE_INDEX (for datastores that support using named indexes). If we had wanted the index to provide uniqueness, we could have made this

```
        <index name="BOOKING_TYPE_INDEX" unique="true"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This has demonstrated indexing the fields of a class. The above example will index together all columns for that field. In certain circumstances you want to be able to index from the column point of view. So we are thinking more from a database perspective. Here we define our indexes at the <class> level, like this

```
<class name="Booking">
    <index name="MY_BOOKING_INDEX">
        <column name="BOOKING"/>
    </index>
    ...
</class>
```

This creates an index for the specified column (where the datastore supports columns i.e RDBMS).

See also :-

- MetaData reference for <index> element
- Annotations reference for @Index
- Annotations reference for @Index (class level)

Unique constraints

Relational Databases (RDBMS) provide the ability to have unique constraints defined on tables to give extra control over data integrity. JDO 2 provides a mechanism for defining such unique constraints. Lets take the previous class, and show how to specify this

```
<class name="Booking">
    <field name="bookingType">
        <unique name="BOOKING_TYPE_CONSTRAINT"/>
    </field>
</class>
```

So in an identical way to the specification of an index. This example specification will result in the column(s) for "bookingType" being enforced as unique in the datastore. In the same way you can specify unique constraints directly to columns - see the example above for indexes.

Again, as for index, you can also specify unique constraints at "class" level in the MetaData file. This is useful to specify where the composite of 2 or more columns or fields are unique. So with this example

```
<class name="Booking">
    <unique name="UNIQUE_PERF">
        <field name="performanceDate"/>
        <field name="startTime"/>
    </unique>

    <field name="performanceDate"/>
    <field name="startTime"/>
</class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The table for Booking has a unique constraint on the columns for the fields performanceDate and startTime

See also :-

- MetaData reference for <unique> element
- Annotations reference for @Unique
- Annotations reference for @Unique (class level)

Foreign Keys

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe ? Lets take an example

```
public class Hotel
{
    private Set rooms;
    ...
}

public class Room
{
    private int numberOfBeds;
    ...
}
```

We now want to control the relationship so that it is linked by a named foreign key, and that we cascade delete the Room object when we delete the Hotel. We define the Meta-Data like this

```
<class name="Hotel">
    <field name="rooms">
        <collection element-type="com.mydomain.samples.hotel.Room"/>
        <foreign-key name="HOTEL_ROOMS_FK" delete-action="cascade"/>
    </field>
</class>
```

So we now have given the datastore control over the cascade deletion strategy for objects stored in these tables. Please be aware that JDO2 provides Dependent Fields as a way of allowing cascade deletion. The difference here is that Dependent Fields is controlled by DataNucleus, whereas foreign key delete actions are controlled by the datastore (assuming the datastore supports it even)

DataNucleus provides an extension that can give significant benefit to users. This is provided via the PersistenceManagerFactory datanucleus.rdbms.constraintCreateMode. This property has 2 values. The default is DataNucleus which will automatically decide which foreign keys are required to satisfy the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

relationships that have been specified, whilst utilising the information provided in the MetaData for foreign keys. The other option is JDO2 which will simply create foreign keys that have been specified in the MetaData file(s).

Note that the *foreign-key* for a 1-N FK relation can be specified as above, or under the *element* element. Note that the *foreign-key* for a 1-N JoinTable relation is specified under *field* for the FK from owner to join table, and is specified under *element* for the FK from join table to element table.

In the special case of application-identity and inheritance there is a foreign-key from subclass to superclass. You can define this as follows

```
<class name="MySubClass">
    <inheritance>
        <join>
            <foreign-key name="ID_FK"/>
        </join>
    </inheritance>
</class>
```

See also :-

- MetaData reference for <foreignkey> element
- Annotations reference for @ForeignKey
- Deletion of related objects using FK constraints

Primary Keys

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. You specify in other parts of the MetaData which fields are part of the primary key (if using applicatioin identity), or you define the name of the column DataNucleus should use for the primary key (if using datastore identity). What these other parts of the MetaData don't allow is specifying the constraint name for the primary key. You can specify this if you wish, like this

```
<class name="Booking">
    <primary-key name="BOOKING_PK"/>
    ...
</class>
```

When the schema is generated for this table, the primary key will be given the specified name, and will use the column(s) specified by the identity type in use.

In the case where you have a 1-N/M-N relation using a join table you can specify the name of the primary key constraint used as follows

```
<class name="Hotel">
    <field name="rooms">
        <collection element-type="com.mydomain.samples.hotel.Room"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <join>
            <primary-key name="HOTEL_ROOM_PK"/>
        </join>
    </field>
</class>
```

This creates a PK constraint with name "HOTEL_ROOM_PK".

See also :-

- MetaData reference for <primary-key> element
- Annotations reference for @PrimaryKey
- Annotations reference for @PrimaryKey (class level)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.9 Inheritance

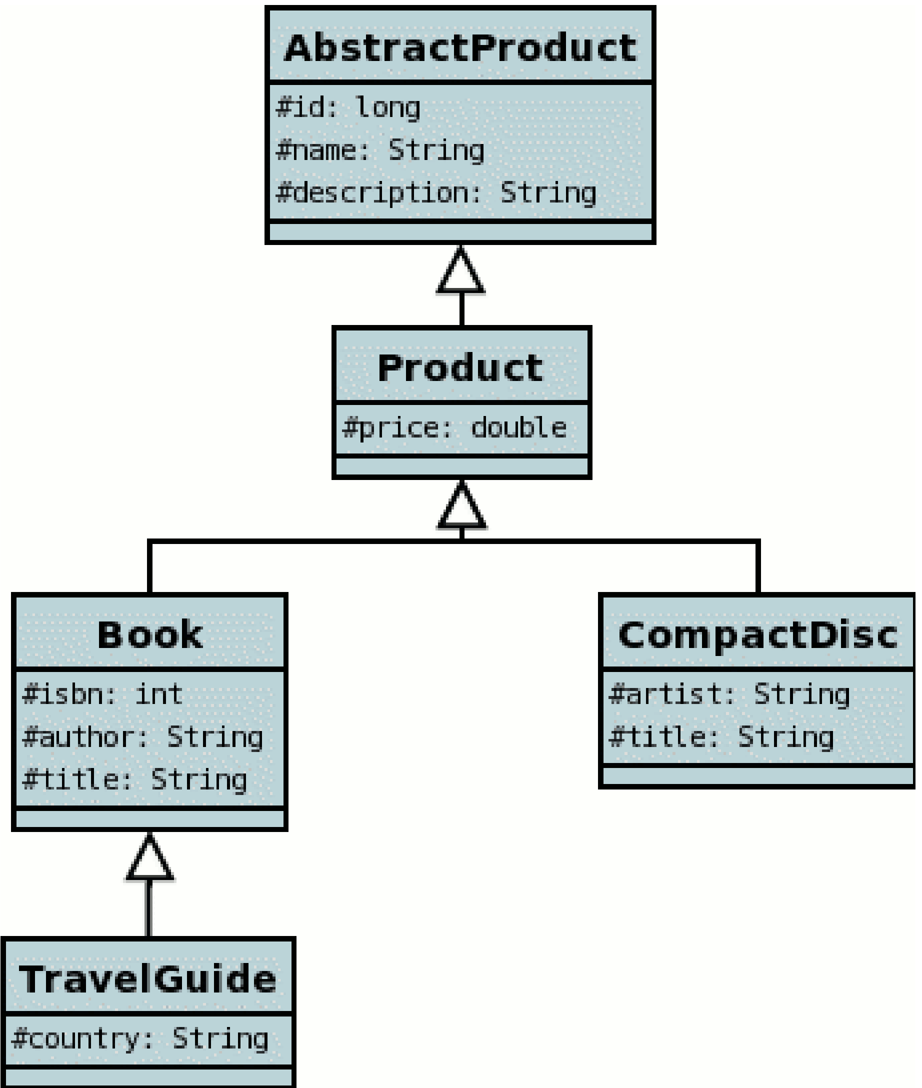
JDO Inheritance Strategies

In Java it is a normal situation to have inheritance between classes. With JDO you have choices to make as to how you want to persist your classes for the inheritance tree. For each class you select how you want to persist that classes information. You have the following choices.

1. The first and simplest to understand option is where each class has its own table in the datastore. In JDO2 this is referred to as new-table.
2. The second way is to select a class to have its fields persisted in the table of its subclass. In JDO2 this is referred to as subclass-table
3. The third way is to select a class to have its fields persisted in the table of its superclass. In JDO2 this is known as superclass-table.
4. A DataNucleus extension way is to have all classes in an inheritance tree with their own table containing all fields. This is known as complete-table and is enabled by setting the inheritance strategy of the root class to use this.

In order to demonstrate the various inheritance strategies we need an example. Here are a few simple classes representing products in a (online) store. We have an abstract base class, extending this to to provide something that we can represent any product by. We then provide a few specialisations for typical products. We will use these classes later when defining how to persistent these objects in the different inheritance strategies.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



JDO2 imposes a "default" inheritance strategy if none is specified for a class. If the class is a base class and no inheritance strategy is specified then it will be set to new-table for that class. If the class has a superclass and no inheritance strategy is specified then it will be set to superclass-table. This means that, when no strategy is set for the classes in an inheritance tree, they will default to using a single table managed by the base class.

You can control the "default" strategy chosen by way of a Extension. This is specified by way of a PMF property datanucleus.defaultInheritanceStrategy. The default is JDO2 which will give the above JDO 2 default behaviour for all classes that have no strategy specified. The other option is TABLE_PER_CLASS which will use "new-table" for all classes which have no strategy specified

See also :-

- MetaData reference for <inheritance> element
- MetaData reference for <discriminator> element
- Annotations reference for @Inheritance
- Annotations reference for @Discriminator

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

New Table

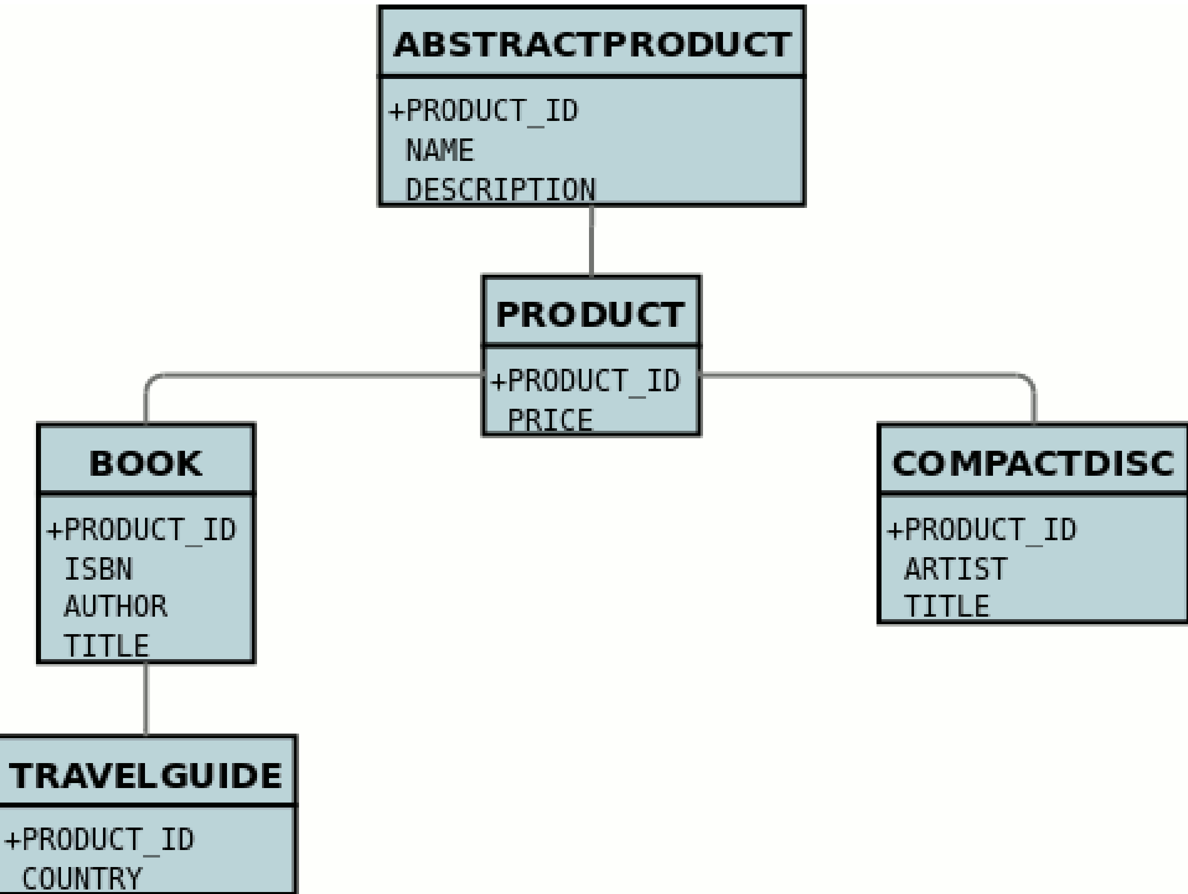
Here we want to have a separate table for each class. This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub type. Let's try an example using the simplest to understand strategy new-table. We have the classes defined above, and we want to persist our classes each in their own table. We define the Meta-Data for our classes like this

```
<class name="AbstractProduct">
    <inheritance strategy="new-table"/>
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="name">
        <column name="NAME"/>
    </field>
    <field name="description">
        <column name="DESCRIPTION"/>
    </field>
</class>
<class name="Product">
    <inheritance strategy="new-table"/>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>
<class name="Book">
    <inheritance strategy="new-table"/>
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
<class name="TravelGuide">
    <inheritance strategy="new-table"/>
    <field name="country">
        <column name="COUNTRY"/>
    </field>
</class>
<class name="CompactDisc">
    <inheritance strategy="new-table"/>
    <field name="artist">
        <column name="ARTIST"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
```

We use the inheritance element to define the persistence of the inherited classes.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

In the datastore, each class in an inheritance tree is represented in its own datastore table (tables ABSTRACTPRODUCT, PRODUCT, BOOK, TRAVELGUIDE, and COMPACTDISC), with the subclasses tables' having foreign keys between the primary key and the primary key of the superclass' table.



In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into ABSTRACTPRODUCT, PRODUCT, BOOK, and TRAVELGUIDE.

Subclass table

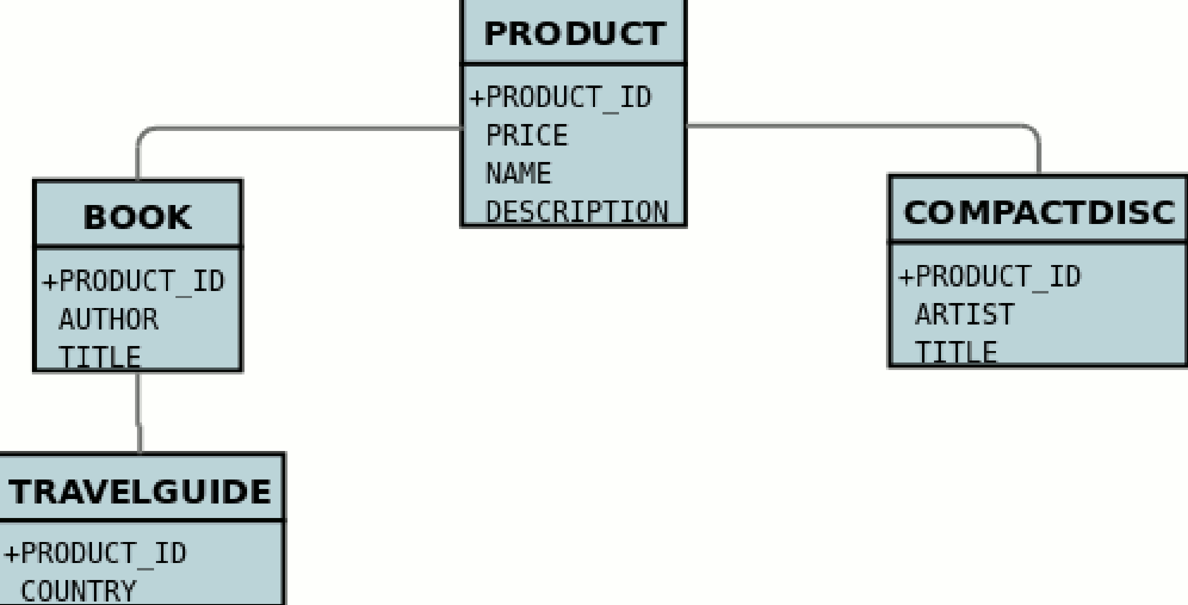
DataNucleus supports persistence of classes in the tables of subclasses where this is required. This is typically used where you have an abstract base class and it doesn't make sense having a separate table for that class. In our example we have no real interest in having a separate table for the AbstractProduct class. So in this case we change one thing in the Meta-Data quoted above. We now change the definition of AbstractProduct as follows

```
<class name="AbstractProduct">
    <inheritance strategy="subclass-table"/>
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="name">
        <column name="NAME"/>
    </field>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="description">
        <column name="DESCRIPTION"/>
    </field>
</class>
```

This subtle change of use the inheritance element has the effect of using the PRODUCT table for both the Product and AbstractProduct classes, containing the fields of both classes.



In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into PRODUCT, BOOK, and TRAVELGUIDE.

DataNucleus doesn't currently support the use of classes defined with subclass-table strategy as having relationships where there are more than a single subclass that has a table. If the class has a single subclass with its own table then there should be no problem.

Superclass table

DataNucleus supports persistence of classes in the tables of superclasses where this is required. This has the advantage that retrieval of an object is a single SQL call to a single table. It also has the disadvantage that the single table can have a very large number of columns, and database readability and performance can suffer, and additionally that a discriminator column is required. In our example, lets ignore the AbstractProduct class for a moment and assume that Product is the base class. We have no real interest in having separate tables for the Book and CompactDisc classes and want everything stored in a single table PRODUCT. We change our MetaData as follows

```
<class name="Product">
    <inheritance strategy="new-table">
        <discriminator strategy="class-name">
            <column name="PRODUCT_TYPE"/>
        </discriminator>
    </inheritance>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>
<class name="Book">
    <inheritance strategy="superclass-table"/>
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
<class name="TravelGuide">
    <inheritance strategy="superclass-table"/>
    <field name="country">
        <column name="COUNTRY"/>
    </field>
</class>
<class name="CompactDisc">
    <inheritance strategy="superclass-table"/>
    <field name="artist">
        <column name="ARTIST"/>
    </field>
    <field name="title">
        <column name="DISCTITLE"/>
    </field>
</class>
```

This change of use of the inheritance element has the effect of using the PRODUCT table for all classes, containing the fields of Product, Book, CompactDisc, and TravelGuide. You will also note that we used a discriminator element for the Product class. The specification above will result in an extra column (called PRODUCT_TYPE) being added to the PRODUCT table, and containing the class name of the object stored. So for a Book it will have "com.mydomain.samples.store.Book" in that column. This column is used in discriminating which row in the database is of which type. The final thing to note is that in our classes Book and CompactDisc we have a field that is identically named. With CompactDisc we have defined that its column will be called DISCTITLE since both of these fields will be persisted into the same table and would have had identical names otherwise - this gets around the problem.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

PRODUCT

+PRODUCT_ID
PRICE
NAME
DESCRIPTION
AUTHOR
TITLE
COUNTRY
ARTIST
DISCTITLE
PRODUCT_TYPE

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into the PRODUCT table only.

JDO 2 allows two types of discriminators. The example above used a discriminator strategy of class-name. This inserts the class name into the discriminator column so that we know what the class of the object really is. The second option is to use a discriminator strategy of value-map. With this we will define a "value" to be stored in this column for each of our classes. The only thing here is that we have to define the "value" in the MetaData for ALL classes that use that strategy. So to give the equivalent example :-

```
<class name="Product">
    <inheritance strategy="new-table">
        <discriminator strategy="value-map" value="PRODUCT">
            <column name="PRODUCT_TYPE"/>
        </discriminator>
    </inheritance>
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>
<class name="Book">
    <inheritance strategy="superclass-table">
        <discriminator value="BOOK"/>
    </inheritance>
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
<class name="TravelGuide">
    <inheritance strategy="superclass-table">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <discriminator value="TRAVELGUIDE"/>
    </inheritance>
    <field name="country">
        <column name="COUNTRY"/>
    </field>
</class>
<class name="CompactDisc">
    <inheritance strategy="superclass-table">
        <discriminator value="COMPACTDISC"/>
    </inheritance>
    <field name="artist">
        <column name="ARTIST"/>
    </field>
    <field name="title">
        <column name="DISCTITLE"/>
    </field>
</class>
```

As you can see from the MetaData DTD it is possible to specify the column details for the discriminator. DataNucleus supports this, but only currently supports the following values of jdbc-type : VARCHAR, CHAR, INTEGER, BIGINT, NUMERIC. The default column type will be a VARCHAR.

Complete table

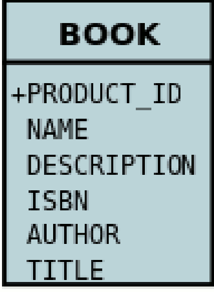
With "complete-table" we define the strategy on the root class of the inheritance tree and it applies to all subclasses. Each class is persisted into its own table, having columns for all fields (of the class in question plus all fields of superclasses). So taking the same classes as used above

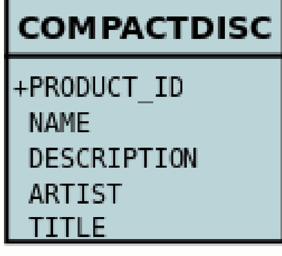
```
<class name="Product">
    <inheritance strategy="complete-table"/>
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>
<class name="Book">
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
<class name="TravelGuide">
    <field name="country">
        <column name="COUNTRY"/>
    </field>
</class>
```

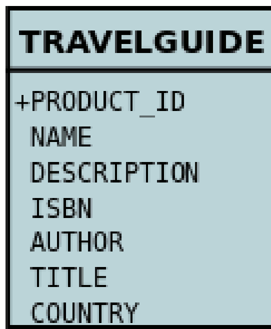
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<class name="CompactDisc">
    <field name="artist">
        <column name="ARTIST"/>
    </field>
    <field name="title">
        <column name="DISCTITLE"/>
    </field>
</class>
```

So the key thing is the specification of inheritance strategy at the root only. This then implies a datastore schema as follows







So any object of explicit type Book is persisted into the table "BOOK". Similarly any TravelGuide is persisted into the table "TRAVELGUIDE". In addition if any class in the inheritance tree is abstract then it won't have a table since there cannot be any instances of that type. **DataNucleus currently has limitations when using a class using this inheritance as the element of a collection.**

Retrieval of inherited objects

JDO provides particular mechanisms for retrieving inheritance trees. These are accessed via the Extent/Query interface. Taking our example above, we can then do

```
tx.begin();
Extent e = pm.getExtent(com.mydomain.samples.store.Product.class, true);
Query  q = pm.newQuery(e);
Collection c=(Collection)q.execute();
tx.commit();
```

The second parameter passed to pm.getExtent relates to whether to return subclasses. So if we pass in the root of the inheritance tree (Product in our case) we get all objects in this inheritance tree returned. You can, of course, use far more elaborate queries using JDOQL, and SQL but this is just to highlight the method of retrieval of subclasses.

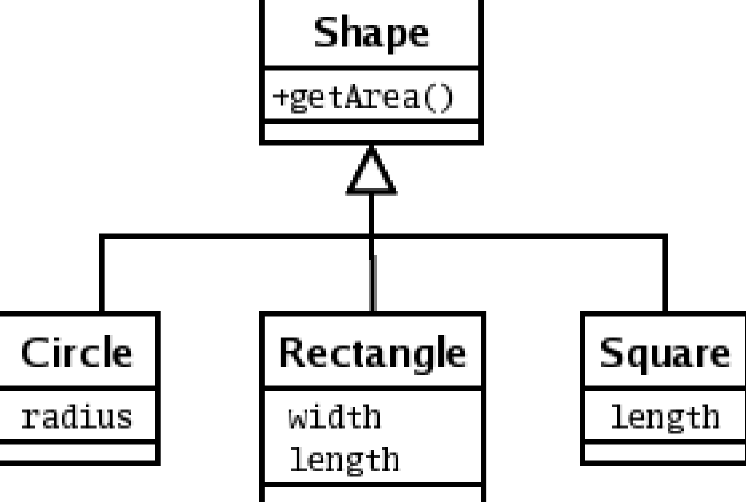
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.10 Interfaces

Interfaces

JDO requires that implementations support the persistence of interfaces as first class objects (FCO's). DataNucleus provides this capability. It follows the same general process as for java.lang.Object since both interfaces and java.lang.Object are basically references to some persistable object.

To demonstrate interface handling lets introduce some classes. Let's suppose you have an interface with a selection of classes implementing the interface something like this



You then have a class that contains an object of this interface type

```
public class ShapeHolder
{
    protected Shape shape=null;
    ...
}
```

JDO doesn't define how an interface is persisted in the datastore. Obviously there can be many implementations and so no obvious solution. DataNucleus allows the following

- **per-implementation** : a FK is created for each implementation so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used
- **identity** : a single column is added and this stores the class name of the implementation stored, as well as the identity of the object. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia** : a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The user controls which one of these is to be used by specifying the *extension* **mapping-strategy** on the field containing the interface. The default is "per-implementation"

1-1

To allow persistence of this interface field with DataNucleus you have 2 levels of control. The first level is global control. Since all of our *Square*, *Circle*, *Rectangle* classes **implements** *Shape* then we just define them in the MetaData as we would normally.

```
<package name="com.mydomain.samples.shape">
    <class name="Square">
        ...
    </class>
    <class name="Circle">
        ...
    </class>
    <class name="Rectangle">
        ...
    </class>
</package>
```

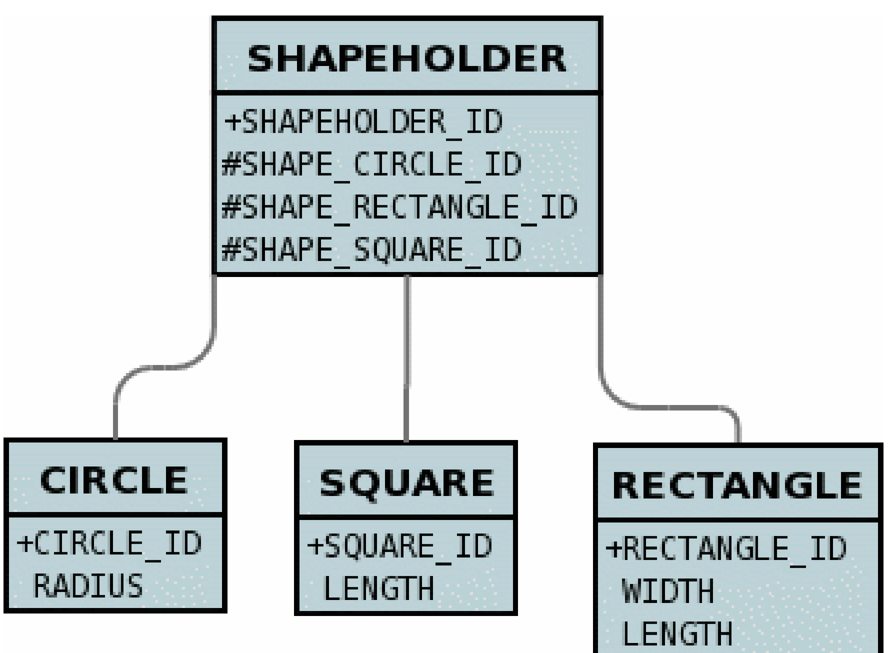
The global way means that when mapping that field DataNucleus will look at all PersistenceCapable classes it knows about that implement the specified interface.

JDO2 also allows users to specify a list of classes implementing the interface on a field-by-field basis, defining which of these implementations are accepted for a particular interface field. To do this you define the Meta-Data like this

```
<package name="com.mydomain.samples.shape">
    <class name="ShapeHolder">
        <field name="shape" persistence-modifier="persistent"
               field-type="com.mydomain.samples.shape.Circle,
                           com.mydomain.samples.shape.Rectangle,
                           com.mydomain.samples.shape.Square"/>
</class>
```

That is, for any interface object in a class to be persisted, you define the possible implementation classes that can be stored there. DataNucleus interprets this information and will map the above example classes to the following in the database

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

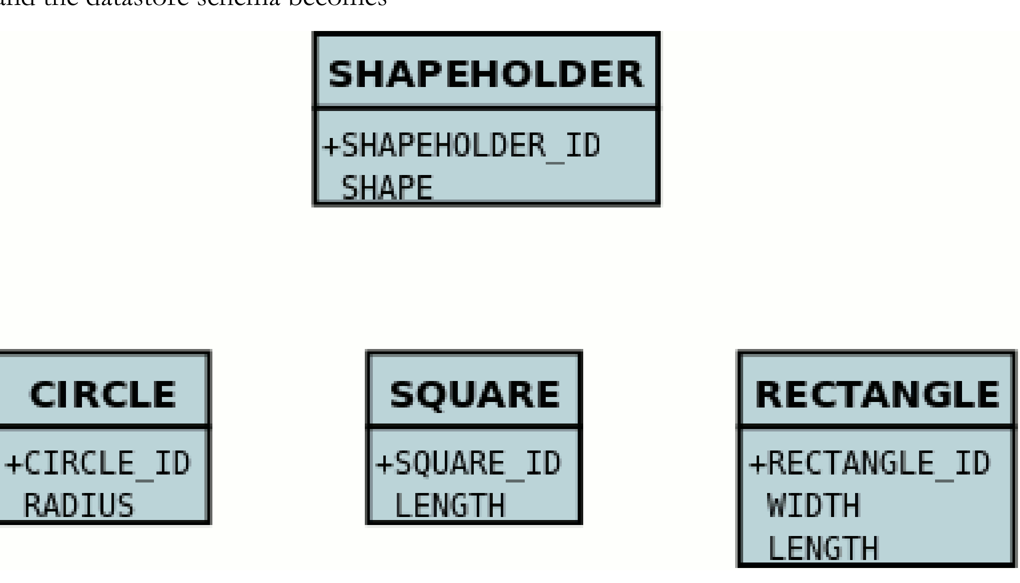


So DataNucleus adds foreign keys from the containers table to all of the possible implementation tables for the shape field.

If we use **mapping-strategy** of "identity" then we get a different datastore schema.

```
    <class name="ShapeHolder">
        <field name="shape" persistence-modifier="persistent">
            <extension vendor-name="datanucleus" key="mapping-strategy"
value="identity"/>
        </field>
    </class>
```

and the datastore schema becomes



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

and the column "SHAPE" will contain strings such as *com.mydomain.samples.shape.Circle:1* allowing retrieval of the related implementation object.

1-N

You can have a Collection/Map containing elements of an interface type. You specify this in the same way as you would any Collection/Map. **You can have a Collection of interfaces as long as you use a join table relation and it is unidirectional.** The "unidirectional" restriction is that the interface is not persistent on its own and so cannot store the reference back to the owner object. You need to use a DataNucleus extension tag "implementation-classes" if you want to restrict the collection to only contain particular implementations of an interface. Use the 1-N relationship guides for the metadata definition to use.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.11 Objects

JDO Objects

JDO requires that implementations support the persistence of java.lang.Object as first class objects (FCO's). DataNucleus provides this capability and also provides that java.lang.Object can be stored as serialised. It follows the same general process as for Interfaces since both interfaces and java.lang.Object are basically references to some persistable object.

JDO doesn't define how an object FCO is persisted in the datastore. Obviously there can be many "implementations" and so no obvious solution. DataNucleus allows the following

- **per-implementation** : a FK is created for each "implementation" so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used
- **identity** : a single column is added and this stores the class name of the "implementation" stored, as well as the identity of the object. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia** : a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the *extension* **mapping-strategy** on the field containing the interface. The default is "per-implementation"

FCO

Let's suppose you have a field in a class and you have a selection of possible persistable class that could be stored there, so you decide to make the field a *java.lang.Object*. So let's take an example. We have the following class

```
public class ParkingSpace
{
    String location;
    Object occupier;
}
```

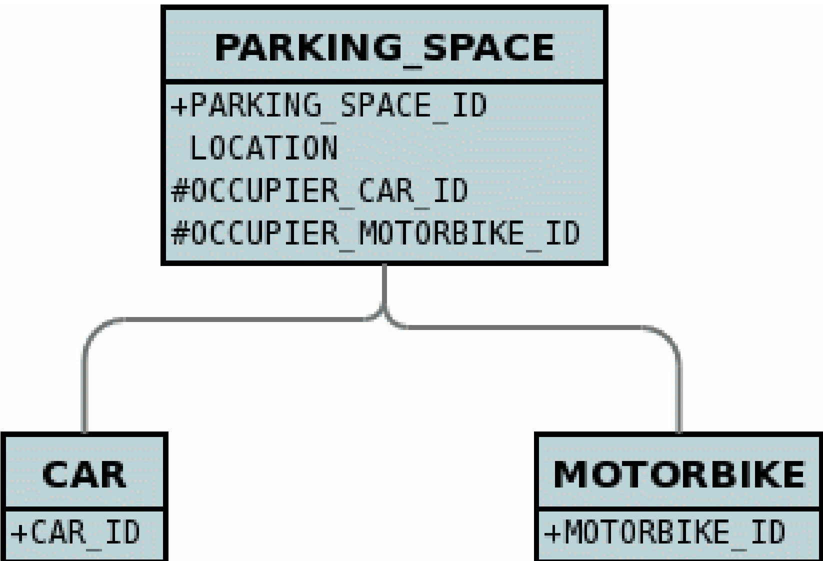
So we have a space in a car park, and in that space we have an occupier of the space. We have some legacy data and so can't make the type of this "occupier" an interface type, so we just use *java.lang.Object*. Now we know that we can only have particular types of objects stored there (since there are only a few types of vehicle that can enter the car park). So we define our MetaData like this

```
<package name="com.mydomain.samples.object">
    <class name="ParkingSpace">
        <field name="location"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <field name="occupier" persistence-modifier="persistent"
               field-type="com.mydomain.samples.vehicles.Car,
                       com.mydomain.samples.vehicles.Motorbike"/>
        </field>
</class>
```

This will result in the following database schema.



So DataNucleus adds foreign keys from the ParkingSpace table to all of the possible implementation tables for the occupier field.

In conclusion, when using "per-implementation" mapping for any java.lang.Object field in a class to be persisted (as non-serialised), you must define the possible "implementation" classes that can be stored there.

If we use **mapping-strategy** of "identity" then we get a different datastore schema.

```
    <class name="ParkingSpace">
        <field name="location"/>
        <field name="occupier" persistence-modifier="persistent">
            <extension vendor-name="datanucleus" key="mapping-strategy"
value="identity"/>
        </field>
    </class>
```

and the datastore schema becomes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

and the column "OCCUPIER" will contain strings such as *com.mydomain.samples.object.Car:1* allowing retrieval of the related implementation object.

Collections of Objects

You can have a Collection/Map containing elements of java.lang.Object. You specify this in the same way as you would any Collection/Map. DataNucleus supports having a Collection of references with multiple implementation types as long as you use a join table relation.

Serialised Objects

By default a field of type java.lang.Object is stored as an instance of the underlying PersistenceCapable in the table of that object. If either your Object field represents non-PersistenceCapable objects or you simply wish to serialise the Object into the same table as the owning object, you need to specify the "serialized" attribute, like this

```
<class name="MyClass">
    <field name="myObject" serialized="true"/>
</class>
```

Similarly, where you have a collection of Objects using a join table, the objects are, by default, stored in the table of the PersistenceCapable instance. If instead you want them to occupy a single BLOB column of the join table, you should specify the "embedded-element" attribute of <collection> like this

```
<class name="MyClass">
    <field name="myCollection">
        <collection element-type="java.lang.Object" serialized-element="true"/>
        <join/>
    </field>
</class>
```

Please refer to the serialised fields guide for more details of storing objects in this way.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

4.12 Arrays

JDO Arrays

JDO allows implementations to optionally support the persistence of arrays. DataNucleus provides full support for arrays in similar ways that collections are supported, but with the proviso that any changes in an array cannot be detected by DataNucleus, and so the whole array field needs updating. DataNucleus supports persisting arrays as

- Single Column - the array is byte-streamed into a single column in the table of the containing object.
- Serialised - the array is serialised into single column in the table of the containing object.
- Using a Join Table - where the array relation is persisted into the join table, with foreign-key links to an element table where the elements of the array are PersistenceCapable
- Using a Foreign-Key in the element - only available where the array is of a PersistenceCapable type

Single Column Arrays

Let's suppose you have a class something like this

Account
firstName: String lastName: String permissions: byte[]

So we have an Account and it has a number of permissions, each expressed as a byte. We want to persist the permissions in a single-column into the table of the account (but we don't want them serialised). We then define MetaData something like this

```
<class name="Account" identity-type="datastore">
    <field name="firstName">
        <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastName">
        <column column="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="permissions" column="PERMISSIONS"/>
</class>
```

You could have added <array> to be explicit but the type of the field is an array, and the type declaration also defines the component type so nothing more is needed. This results in a datastore schema as follows

DataNucleus supports persistence of the following array types in this way : *boolean[], byte[], char[], double[], float[], int[], long[], short[], Boolean[], Byte[], Character[], Double[], Float[], Integer[], Long[], Short[], BigDecimal[], BigInteger[]*

See also :-

- MetaData reference for <array> element
- Annotations reference for @Element

Serialised Arrays

Let's suppose you have a class something like this

Account
firstName: String
lastName: String
permissions: byte[]

So we have an Account and it has a number of permissions, each expressed as a byte. We want to persist the permissions as serialised into the table of the account. We then define MetaData something like this

```
<class name="Account" identity-type="datastore">
    <field name="firstName">
        <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastName">
        <column column="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="permissions" serialized="true" column="PERMISSIONS"/>
</class>
```

That is, you define the field as serialized. To define arrays of short, long, int, or indeed any other supported array type you would do the same as above. This results in a datastore schema as follows

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

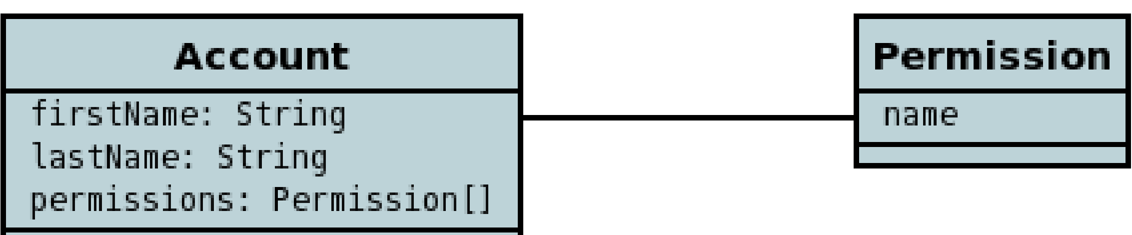
DataNucleus supports persistence of many array types in this way, including : *boolean[], byte[], char[], double[], float[], int[], long[], short[], Boolean[], Byte[], Character[], Double[], Float[], Integer[], Long[], Short[], BigDecimal[], BigInteger[], String[], java.util.Date[], java.util.Locale[]*

See also :-

- MetaData reference for <field> element
- MetaData reference for <array> element
- Annotations reference for @Persistent
- Annotations reference for @Element
- Annotations reference for @Serialized

Arrays persisted into Join Tables

DataNucleus will support arrays persisted into a join table. Let's take the example above and make the "permission" a class in its own right, so we have



So an Account has an array of Permissions, and both of these objects are PersistenceCapable. We want to persist the relationship using a join table. We define the MetaData as follows

```
<class name="Account" table="ACCOUNT">
    <field name="firstName">
        <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastName">
        <column column="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="permissions" table="ACCOUNT_PERMISSIONS">
        <array/>
        <join column="ACCOUNT_ID"/>
        <element column="PERMISSION_ID"/>
        <order column="PERMISSION_ORDER_IDX"/>
    </field>
</class>
<class name="Permission" table="PERMISSION">
    <field name="name"/>
```

```
</class>
```

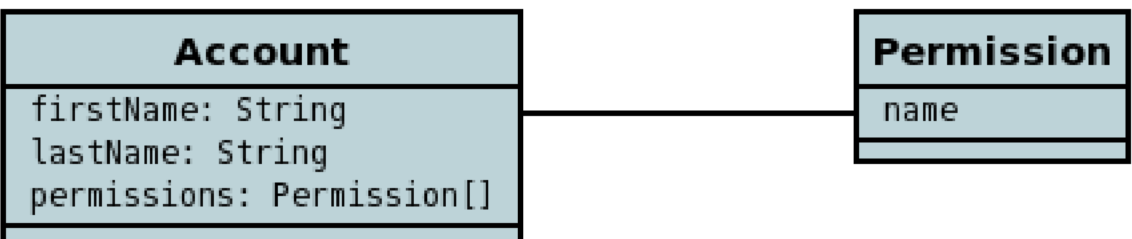
This results in a datastore schema as follows

See also :-

- MetaData reference for <array> element
- MetaData reference for <element> element
- MetaData reference for <join> element
- MetaData reference for <order> element
- Annotations reference for @Element
- Annotations reference for @Order

Arrays persisted using Foreign-Keys

DataNucleus will support arrays persisted via a foreign-key in the element table. This is only applicable when the array is of a PersistenceCapable type. Let's take the same example above. So we have



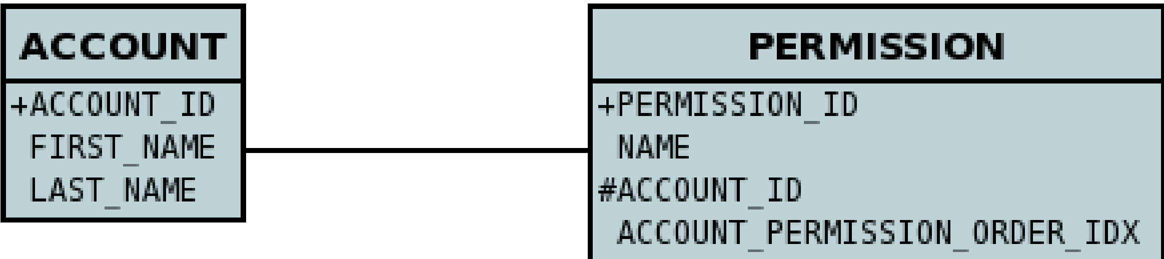
So an Account has an array of Permissions, and both of these objects are PersistenceCapable. We want to persist the relationship using a foreign-key in the table for the Permission class. We define the MetaData as follows

```
<class name="Account" table="ACCOUNT">
    <field name="firstName">
        <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastName">
        <column column="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="permissions">
        <array/>
        <element column="ACCOUNT_ID"/>
        <order column="ACCOUNT_PERMISSION_ORDER_IDX"/>
    </field>
</class>
<class name="Permission" table="PERMISSION">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <field name="name"/>
</class>
```

This results in a datastore schema as follows



See also :-

- MetaData reference for <array> element
- MetaData reference for <element> element
- MetaData reference for <order> element
- Annotations reference for @Element
- Annotations reference for @Order

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4.13 Versioning

JDO Versioning

JDO2 allows objects of classes to be versioned. The version is typically used as a way of detecting if the object has been updated by another thread or PersistenceManager since retrieval using the current PersistenceManager - for use by Optimistic Transactions. JDO defines several "strategies" for generating the version of an object. The strategy has the following possible values

- **none** stores a number like the version-number but will not perform any optimistic checks.
- **version-number** stores a number (starting at 1) representing the version of the object.
- **date-time** stores a Timestamp representing the time at which the object was last updated. *Note that not all RDBMS store milliseconds in a Timestamp!*
- **state-image** stores a Long value being the hash code of all fields of the object. **DataNucleus doesnt currently support this option**

Versioning using a surrogate column

JDO2s mechanism for versioning of objects in RDBMS datastores is via a **surrogate column** in the table of the class. In the MetaData you specify the details of the surrogate column and the strategy to be used. For example

```
<package name="mydomain">
    <class name="User" table="USER">
        <version strategy="version-number" column="VERSION"/>
        <field name="name" column="NAME"/>
        ...
    </class>
</package>
```

alternatively using annotations

```
@PersistenceCapable
@Version(strategy="version-number", column="VERSION")
public class MyClass
{
    ...
}
```

The specification above will create a table with an additional column called "VERSION" that will store the version of the object.

Versioning using a field of the class

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DataNucleus provides a valuable extension to JDO whereby you can have a field of your class store the version of the object. This equates to JPA's versioning process whereby you have to have a field present. To do this lets take a class

```
public class User
{
    String name;
    ...
    long myVersion;

}
```

and we want to store the version of the object in the field "myVersion". So we specify the metadata as follows

```
<package name="mydomain">
    <class name="User" table="USER">
        <version strategy="version-number">
            <extension vendor-name="datanucleus" key="field-name"
value="myVersion"/>
        </version>
        <field name="name" column="NAME"/>
        ...
        <field name="myVersion" column="VERSION"/>
    </class>
</package>
```

alternatively using annotations

```
@PersistenceCapable
@Version(strategy="version-number", column="VERSION",
         extensions={@Extension(vendorName="datanucleus", key="field-name",
value="myVersion")})
public class MyClass
{
    protected long myVersion;
    ...
}
```

and so now objects of our class will have access to the version via the "myVersion" field.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.1 Managing Relationships

Relationships

The power of a Java persistence solution like DataNucleus is demonstrated when persisting relationships between objects. There are many types of relationships.

- 1-1 relationships - this is where you have an object A relates to a second object B. The relation can be *unidirectional* where A knows about B, but B doesnt know about A. The relation can be *bidirectional* where A knows about B and B knows about A.
- 1-N relationships - this is where you have an object A that has a collection of other objects of type B. The relation can be *unidirectional* where A knows about the objects B but the Bs dont know about A. The relation can be *bidirectional* where A knows about the objects B and the Bs know about A
- N-1 relationships - this is where you have an object B1 that relates to an object A, and an object B2 that relates to A also etc. The relation can be *unidirectional* where the A doesnt know about the Bs. The relation can be *bidirectional* where the A has a collection of the Bs. [i.e a 1-N relationship but from the point of view of the element]
- M-N relationships - this is where you have objects of type A that have a collection of objects of type B and the objects of type B also have a collection of objects of type A. The relation is always *bidirectional* by definition
- Compound Identity relationships when you have a relation and part of the primary key of the related object is the other persistent object. This is only available in JDO

Assigning Relationships

When the relation is *unidirectional* you simply set the related field to refer to the other object. For example we have classes A and B and the class A has a field of type B. So we set it like this

```
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
```

When the relation is *bidirectional* you **have to set both sides** of the relation. For example, we have classes A and B and the class A has a collection of elements of type B, and B has a field of type A. So we set it like this

```
A a = new A();
B b1 = new B();
a.addElement(b1); // "a" knows about "b1"
b1.setA(a); // "b1" knows about "a"
```

So it is really simple, with only 1 general rule. **With a *bidirectional* relation you should set both sides of the relation**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Managed Relationships

As previously mentioned, users should really set both sides of a bidirectional relation. DataNucleus provides a good level of **managed relations** in that it will attempt to correct any missing information in relations to make both sides consistent. This is defined below

For a **1-1 bidirectional relation**, at persist you should set one side of the relation and the other side will be set to make it consistent. If the respective sides are set to inconsistent objects then an exception will be thrown at persist. At update of owner/non-owner side the other side will also be updated to make them consistent.

For a **1-N bidirectional relation** and you only specify the element owner then the collection must be Set-based since DataNucleus cannot generate indexing information for you in that situation (you must position the elements). At update of element or owner the other side will also be updated to make them consistent. At delete of element the owner collection will also be updated to make them consistent. **If you are using a List you MUST set both sides of the relation**

For an **M-N bidirectional relation**, at persist you MUST set the one side and the other side will be populated at commit/flush to make them consistent.

This management of relations can be turned on/off using a PMF property *datanucleus.manageRelationships*. If you always set both sides of a relation at persist or update then you could safely turn it off.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.2 Cascading

Cascading Operations

When defining your objects to be persisted and the relationships between them, it is often required to define dependencies between these related objects. What should happen when persisting an object and it relates to another object? What should happen to a related object when an object is deleted? You can define what happens with JDO2 and with DataNucleus. Let's take an example

```
public class Owner
{
    private DrivingLicense license;
    private Collection cars;

    ...
}

public class DrivingLicense
{
    private String serialNumber;

    ...
}

public class Car
{
    private String registrationNumber;
    private Owner owner;

    ...
}
```

So we have an *Owner* of a collection of vintage *Car*'s (1-N), and the *Owner* has a *DrivingLicense* (1-1). We want to define lifecycle dependencies to match the relationships that we have between these objects. Firstly lets look at the basic Meta-Data for the objects.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="com.mydomain.samples.cars">
        <class name="Owner">
            <field name="license" persistence-modifier="persistent"/>
            <field name="cars">
                <collection element-type="com.mydomain.samples.cars.Car"
mapped-by="owner"/>
            </field>
        </class>

        <class name="DrivingLicense">
                <field name="serialNumber"/>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <class name="Car">
            <field name="registrationNumber"/>
            <field name="owner" persistence-modifier="persistent"/>
        </class>
    </package>
</jdo>
```

Persistence

JDO2 defines a concept called **persistence-by-reachability**. This means that when you persist an object and it has a related persistable object then this other object is also persisted. So using our example if we do

```
Owner bob = new Owner("Bob Smith");
DrivingLicense license = new DrivingLicense("011234BX4J");
bob.setLicense(license);
pm.makePersistent(bob); // "bob" knows about "license"
```

This results in both the *Owner* and the *DrivingLicense* objects being made persistent since the *Owner* is passed to the PM operation and it has a field referring to the unpersisted *DrivingLicense* object. So "reachability" will persist the license.

With DataNucleus you can actually turn off *persistence-by-reachability* for particular fields, by specifying in the MetaData a DataNucleus extension tag, as follows

```
<class name="Owner">
    <field name="license" persistence-modifier="persistent">
        <extension vendor-name="datanucleus" key="cascade-persist" value="false"/>
    </field>
    ...
</class>
```

So with this specification when we call *makePersistent()* with an object of type *Owner* then the field "license" will not be persisted at that time.

Update

As mentioned above JDO2 defines a concept called **persistence-by-reachability**. This applies not just to persist but also to update of objects, so when you update an object and its updated field has a persistable object then that will be persisted. So using our example if we do

```
Owner bob = (Owner)pm.getObjectById(id);
DrivingLicense license2 = new DrivingLicense("233424BX4J");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
bob.setLicense(license2); // "bob" knows about "license2"
```

So when this field is updated the new *DrivingLicense* object will be made persistent since it is reachable from the persistent *Owner* object.

With DataNucleus you can actually turn off *update-by-reachability* for particular fields, by specifying in the MetaData a DataNucleus extension tag, as follows

```
<class name="Owner">
    <field name="license" persistence-modifier="persistent">
        <extension vendor-name="datanucleus" key="cascade-update" value="false"/>
    </field>
    ...
</class>
```

So with this specification when we call *makePersistent()* to update an object of type Owner then the field "license" will not be updated at that time.

Deletion, using Dependent Field

So we have an inverse 1-N relationship (no join table) between our *Owner* and his precious *Car*'s, and a 1-1 relationship between the *Owner* and his *DrivingLicense*, because without his license he wouldn't be able to drive the cars :-0. What will happen to the *license* and the *cars* when the *owner* dies ? Well in this particular case we want to define that the when the *owner* is deleted, then his *license* will also be deleted (since it is for him only), but that his *cars* will continue to exist, because his daughter will inherit them. In JDO2 this is called Dependent Fields. To utilise this concept to achieve our end goal we change the Meta-Data to be

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="com.mydomain.samples.cars">
        <class name="Owner">
            <field name="license" persistence-modifier="persistent"
dependent="true"/>
            <field name="cars">
                <collection element-type="com.mydomain.samples.cars.Car"
mapped-by="owner"
                    dependent-element="false"/>
            </field>
        </class>

        <class name="DrivingLicense">
            <field name="serialNumber"/>
        </class>

        <class name="Car">
            <field name="registrationNumber"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <field name="owner" persistence-modifier="persistent"
dependent="false"/>
        </class>
    </package>
</jdo>
```

So it was as simple as just adding dependent and dependent-element attributes to our related fields. Notice that we also added one to the other end of the Owner-Car relationship, so that when a *Car* comes to the end of its life, the *Owner* will not die with it. It may be the case that the owner dies driving the car and they both die at the same time, but their deaths are independent!!

Dependent Fields is utilised in the following situations

- An object is deleted (using *deletePersistent()*) and that object has relations to other objects. If the other objects (either 1-1, 1-N, or M-N) are dependent then they are also deleted.
- An object has a 1-1 relation with another object, but the other object relation is nulled out. If the other object is dependent then it is deleted when the relation is nulled.
- An object has a 1-N collection relation with other objects and the element is removed from the collection. If the element is dependent then it will be deleted when removed from the collection. The same happens when the collections is cleared.
- An object has a 1-N map relation with other objects and the key is removed from the map. If the key or value are dependent and they are not present in the map more than once they will be deleted when they are removed. The same happens when the map is cleared.

Deletion, using Foreign Keys (RDBMS)

With JDO2 you can use "dependent-field" as shown above. As an alternative, when using RDBMS, you can use the datastore-defined foreign keys and let the datastore built-in "referential integrity" look after such deletions. DataNucleus provides a PMF property *datanucleus.deletionPolicy* allowing enabling of this mode of operation.

The default setting of *datanucleus.deletionPolicy* is "JDO2" which performs deletion of related objects as follows

1. If *dependent-field* is true then use that to define the related objects to be deleted.
2. Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone
3. Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

The other setting of *datanucleus.deletionPolicy* is "DataNucleus" which performs deletion of related objects as follows

1. If *dependent-field* is true then use that to define the related objects to be deleted.
2. If a *foreign-key* is specified (in MetaData) for the relation field then leave any deletion to the datastore to perform (or throw exceptions as necessary)
3. Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

4. Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

So, as you can see, with the second option you have the ability to utilise datastore "referential integrity" checking using your MetaData-specified <foreign-key> elements.

Persisting Relationships - Reachability At Commit

One further complication is that with JDO there is also a process called **persistence-by-reachability at commit**. When objects are persisted, other objects are persisted with them. If some relations are changed *before* commit and some of these related objects are no longer required to be persistent then they will not be persisted. For example, using our classes above

```
Owner bob = new Owner("Bob Smith");
DrivingLicense license = new DrivingLicense("233424BX4J");
bob.setLicense(license); // "bob" knows about "license"
pm.makePersistent(bob);

DrivingLicense license2 = new DrivingLicense("344566A99XH");
bob.setLicense(license2); // "bob" doesnt know about "license" now. It knows about
"license2" now.

// "bob" and "license2" will be persisted but "license" wont be since not persisted
explicitly
// and at commit it is no longer reachable from a persisted object
tx.commit();
```

With DataNucleus you can turn off **persistence-by-reachability at commit** by setting the *PersistenceManagerFactory* property *datanucleus.persistenceByReachabilityAtCommit* to false.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.3 1-to-1

JDO 1-1 Relationships

You have a 1-to-1 relationship when an object of a class has an associated object of another class (only one associated object). It could also be between an object of a class and another object of the same class (obviously). You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only one of the classes knows about the other class (unidirectional). These are described below.

The various possible relationships are described below.

- 1-1 Unidirectional (where only 1 object is aware of the other)
- 1-1 Bidirectional (where both objects are aware of each other)
- 1-1 Unidirectional "Compound Identity" (object as part of PK in other object)

Unidirectional

For this case you could have 2 classes, User and Account, as below.
so the Account class knows about the User class, but not vice-versa. If you define the Meta-Data for these classes as follows

```
<package name="mydomain">
    <class name="User" table="USER">
        <field name="id" primary-key="true">
            <column name="USER_ID"/>
        </field>
        <field name="login">
            <column name="LOGIN" length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>

    <class name="Account" table="ACCOUNT">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="secondName">
            <column name="SECONDNAME" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="user" persistence-modifier="persistent">
            <column name="USER_ID"/>
        </field>
    </class>
</package>
```

This will create 2 tables in the database, one for User (with name USER), and one for Account (with

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

name ACCOUNT and a column USER_ID), as shown below.

Things to note :-

- Account has the object reference (and so owns the relation) to User and so its table holds the foreign-key
- If you call PM.deletePersistent() on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.

Bidirectional

For this case you could have 2 classes, User and Account again, but this time as below. Here the Account class knows about the User class, and also vice-versa.

Here we create the 1-1 relationship with a single foreign-key. To do this you define the MetaData as

```
<package name="mydomain">
    <class name="User" table="USER">
        <field name="id" primary-key="true">
            <column name="USER_ID"/>
        </field>
        <field name="login">
            <column name="LOGIN" length="20" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent" mapped-by="user">
        </field>
    </class>

    <class name="Account" table="ACCOUNT">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="secondName">
            <column name="SECONDNAME" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="user" persistence-modifier="persistent">
            <column name="USER_ID"/>
        </field>
    </class>
</package>
```

The difference is that we added mapped-by to the field of User. This will create 2 tables in the database, one for User (with name USER), and one for Account (with name ACCOUNT including a USER_ID). The fact that we specified the mapped-by on the User class means that the foreign-key is created in the ACCOUNT table.

Things to note :-

- The "mapped-by" is specified on User (the non-owning side) and so the foreign-key is held by the table of Account (the owner of the relation)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Embedded

The above 2 relationship types assume that both classes in the 1-1 relation will have their own table. You can, of course, embed the elements of one class into the table of the other. This is described in Embedded PC Objects.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.4 1-to-N

JDO 1-N Relationships

You have a 1-N (one to many) when you have one object of a class that has a Collection/Map of objects of another class. In the *java.util* package there are an assortment of possible collection/map classes and they all have subtly different behaviour with respect to allowing nulls, allowing duplicates, providing ordering, etc. There are two ways in which you can represent a collection or map in a datastore : Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the collection or map.

We split our documentation based on what type of collection/map you are using.

- 1-N using Collection types
- 1-N using Set types
- 1-N using List type
- 1-N using Map type

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.4.1 1-to-N (Collection)

JDO 1-N/N-1 Relationships with Collections

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Collection of objects of another class. Please note that Collections allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore : Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Collection.

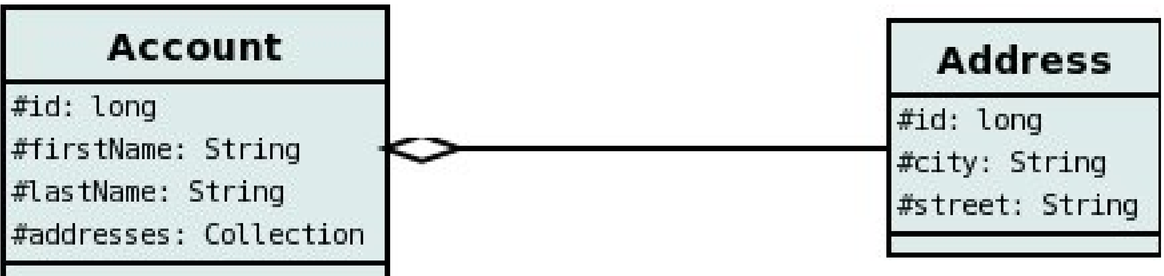
The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised collection
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

Important : If you declare a field as a Collection, you can instantiate it as either Set-based or as List-based. With a List an "ordering" column is required, whereas with a Set it isn't. Consequently DataNucleus needs to know if you plan on using it as Set-based or List-based. You do this by adding an "order" element to the field if it is to be instantiated as a List-based collection. If there is no "order" element, then it will be assumed to be Set-based.

1-N Collection Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
table="ACCOUNT_ADDRESSES">
            <collection element-type="com.mydomain.Address"/>
            <join column="ACCOUNT_ID_OID"/>
            <element column="ADDRESS_ID_EID"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an <order> element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).
- If you want the set to include nulls, you can turn on this behaviour by adding the DataNucleus extension metadata "allows-null" to the <field> set to true

Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

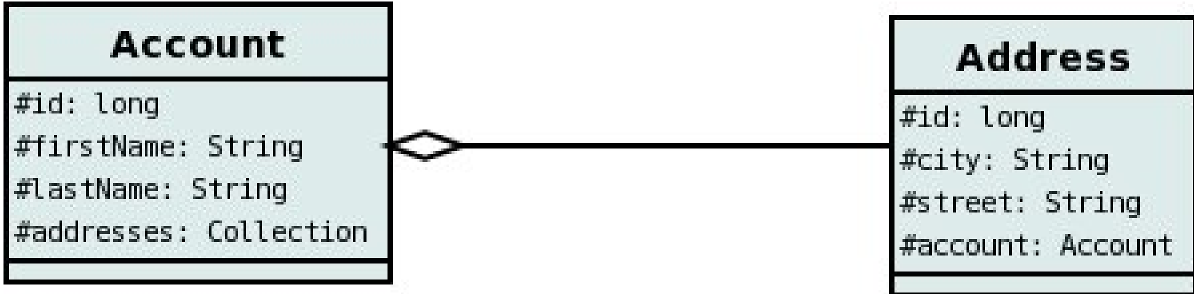
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

1-N Collection Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

control this by adding an <order> element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).

- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allows-null" to the <field> set to true

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
            <column name="ACCOUNT_ID"/>
        </field>
    </class>
</package>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JDO implementation to look for a field called account on the Address class. This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

1-N Collection of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the Meta-Data like this

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="java.lang.String"/>
            <join/>
            <element column="ADDRESS"/>
        </field>
    </class>
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String.

Please note that the column ADPT_PK_IDX is added by DataNucleus so that duplicates can be stored. You can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" within the <field> for the Collection.

Embedded into a Join Table

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

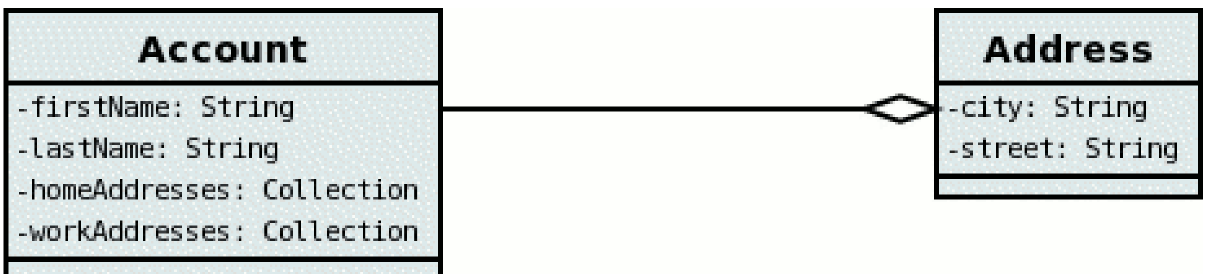
The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.

Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the serialised-element attribute on the collection MetaData element. This is described in Serialised Collection Elements

Shared Join Tables

The relationships using join tables shown above rely on the join table relating to the relation in question. DataNucleus allows the possibility of sharing a join table between relations. The example below demonstrates this. We take the example as show above (1-N Unidirectional Join table relation), and extend Account to have 2 collections of Address records. One for home addresses and one for work addresses, like this



We now change the metadata we had earlier to allow for 2 collections, but sharing the join table

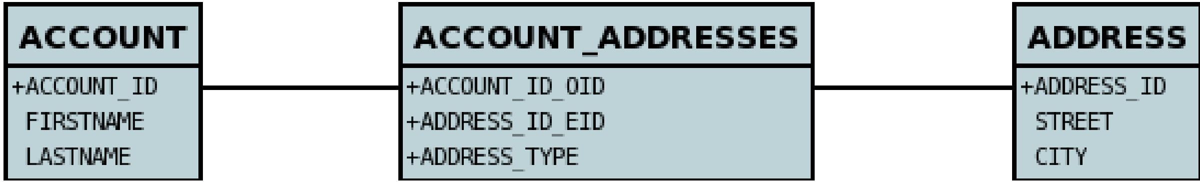
```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="workAddresses" persistence-modifier="persistent"
table="ACCOUNT_ADDRESSES">
            <collection element-type="com.mydomain.Address"/>
            <join column="ACCOUNT_ID_OID"/>
            <element column="ADDRESS_ID_EID"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-column"
value="ADDRESS_TYPE"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-pk"
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
value="true"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-value"
value="work"/>
        </field>
        <field name="homeAddresses" persistence-modifier="persistent"
table="ACCOUNT_ADDRESSES">
            <collection element-type="com.mydomain.Address"/>
            <join column="ACCOUNT_ID_OID"/>
            <element column="ADDRESS_ID_EID"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-column"
value="ADDRESS_TYPE"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-pk"
value="true"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-value"
value="home"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

So we have defined the same join table for the 2 collections "ACCOUNT_ADDRESSES", and the same columns in the join table, meaning that we will be sharing the same join table to represent both relations. The important step is then to define the 3 DataNucleus *extension* tags. These define a column in the join table (the same for both relations), and the value that will be populated when a row of that collection is inserted into the join table. In our case, all "home" addresses will have a value of "home" inserted into this column, and all "work" addresses will have "work" inserted. This means we can now identify easily which join table entry represents which relation field.

This results in the following database schema

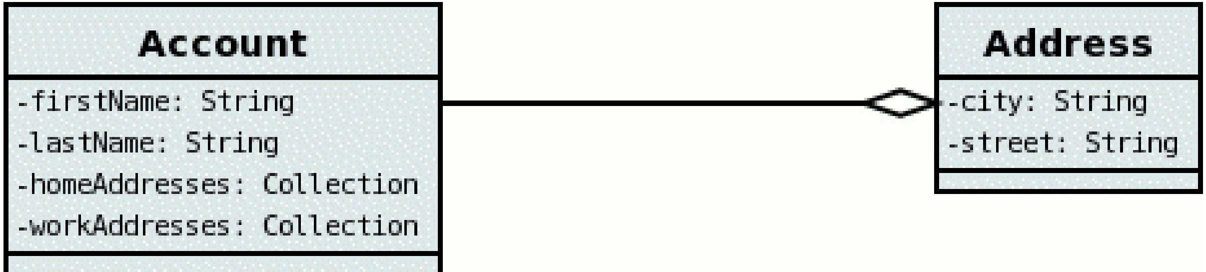


Shared Foreign Key

The relationships using foreign keys shown above rely on the foreign key relating to the relation in question. DataNucleus allows the possibility of sharing a foreign key between relations between the same classes. The example below demonstrates this. We take the example as show above (1-N Unidirectional

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Foreign Key relation), and extend Account to have 2 collections of Address records. One for home addresses and one for work addresses, like this



We now change the metadata we had earlier to allow for 2 collections, but sharing the join table

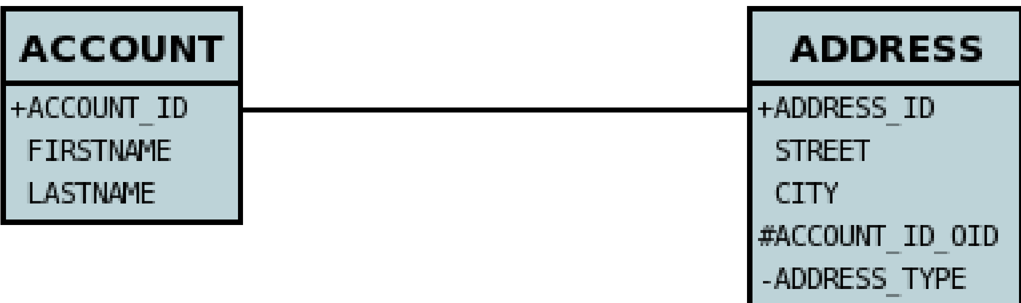
```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="workAddresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID_OID"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-column"
value="ADDRESS_TYPE"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-value"
value="work"/>
        </field>
        <field name="homeAddresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID_OID"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-column"
value="ADDRESS_TYPE"/>
            <extension vendor-name="datanucleus" key="relation-discriminator-value"
value="home"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

So we have defined the same foreign key for the 2 collections "ACCOUNT_ID_OID", The important

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

step is then to define the 2 DataNucleus *extension* tags. These define a column in the element table (the same for both relations), and the value that will be populated when a row of that collection is inserted into the element table. In our case, all "home" addresses will have a value of "home" inserted into this column, and all "work" addresses will have "work" inserted. This means we can now identify easily which element table entry represents which relation field.

This results in the following database schema



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.4.2 1-to-N (Set)

JDO 1-N/N-1 Relationships with Sets

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Set of objects of another class. Please note that Sets do not allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore : Join Table (where e join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Set.

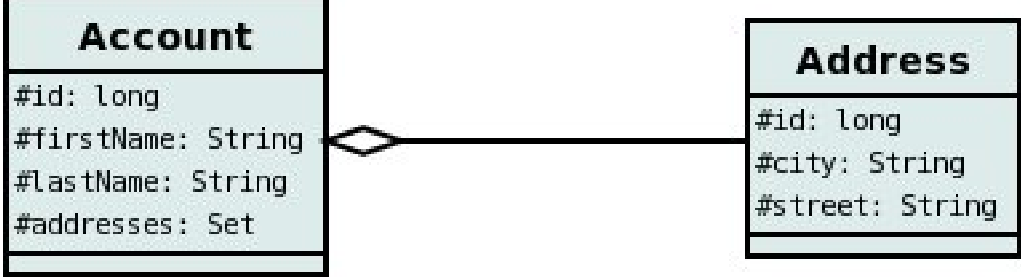
The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised Set
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

This page is aimed at Set fields and so applies to fields of Java type *java.util.HashSet, java.util.LinkedHashSet, java.util.Set, java.util.SortedSet, java.util.TreeSet*

1-N Set Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table"

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

extension

- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allows-null" to the <field> set to true

Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

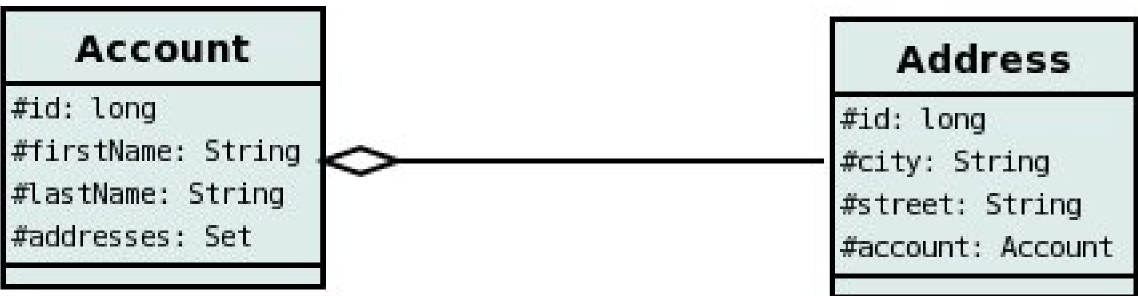
you should use the "Join Table" relationship above.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.

1-N Set Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allows-null" to the <field> set to true

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the Meta-Data for these classes as follows

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
        </field>
    </class>

    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        <field name="city">
            <column name="CITY" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street">
            <column name="STREET" length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
            <column name="ACCOUNT_ID"/>
        </field>
    </class>
</package>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JDO implementation to look for a field called account on the Address class. This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

1-N Set of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the Meta-Data like this

```
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="firstName">
            <column name="FIRSTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName">
            <column name="LASTNAME" length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="java.lang.String"/>
            <join/>
            <element column="ADDRESS"/>
        </field>
    </class>
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the element is not valid to be part of a primary key (with the RDBMS being used). If the column type of your element is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column. You can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" within the <field> for the Collection.

Embedded into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.

Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the serialised-element attribute on the collection MetaData element. This is described in Serialised Collection Elements

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.4.3 1-to-N (List)

JDO 1-N/N-1 Relationships with Lists

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a List of objects of another class. There are two ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the List.

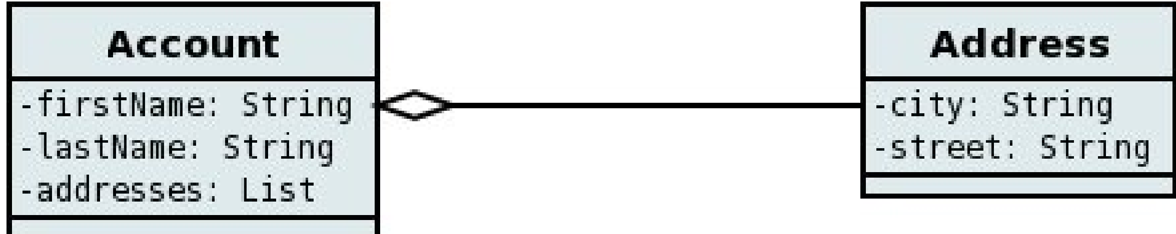
The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Ordered List using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised List
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

This page is aimed at List fields and so applies to fields of Java type *java.util.ArrayList, java.util.LinkedList, java.util.List, java.util.Stack, java.util.Vector*

1-N List Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. There will be 3 tables, one for Address, one for Account, and the join table. The difference from Set is in the contents of the join table. An index column (INTEGER_IDX) is added to keep track of the position of objects in the List. The name of this column can be controlled using the <order> MetaData element.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element and order elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an <order> element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the <field> set to true

Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

element

- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.

Limitations

- Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the List. If you want to allow duplicate List entries, then use the "Join Table" variant above.

1-N Ordered List using Foreign-Key

This is the same as the case above except that we don't want an indexing column adding to the element and instead we define an "ordering" criteria. This is a DataNucleus extension to JDO. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="com.mydomain.Address"/>
            <order>
                <extension vendor-name="datanucleus" key="list-ordering" value="city
ASC"/>
            </order>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

As above there will be 2 tables, one for Address, and one for Account. We have no indexing column, but instead we will order the elements using the "city" field in ascending order.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

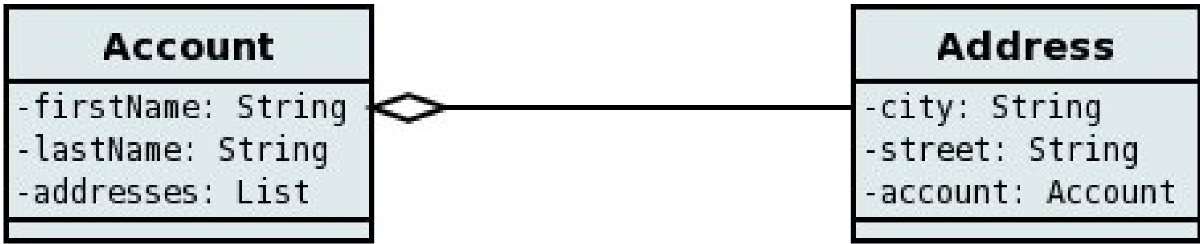
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Limitations

- Ordered lists are only ordered in the defined way **when retrieved** from the datastore.

1-N List Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
        </field>
    </class>
</package>
```

The crucial part is the join element on the field element - this signals to JDO to use a join table. There will be 3 tables, one for Address, one for Account, and the join table. The difference from Set is in the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

contents of the join table. An index column (INTEGER_IDX) is added to keep track of the position of objects in the List. The name of this column can be controlled using the <order> MetaData element.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element and order elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an <order> element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the <field> set to true

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

Please note that an Foreign-Key List will NOT, by default, allow duplicates. This is because it stores the element position in the element table. If you need a List with duplicates we recommend that you use the Join Table List implementation above. If you have an application identity element class then you could in principle add the element position to the primary key to allow duplicates, but this would imply changing your element class identity.

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
            <column name="ACCOUNT_ID"/>
        </field>
    </class>
</package>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JDO implementation to look for a field called account on the Address class. Again there will be 2 tables, one for Address, and one for Account. The difference from the Set example is that the List index is placed in the table for Address whereas for a Set this is not needed.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field (you can't just take the Address object and set its Account field since the position of the Address in the List needs setting, and this is done by adding the Address to the Account). In addition, if you are removing an object from a List, you cannot simply set the owner on the element to "null". You have to remove it from the List end of the relationship.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the List. If you want to allow duplicate List entries, then use the "Join Table" variant above.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1-N List of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a List of primitive or Object types. For example, when you have a List of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the list elements. Let's take our example. We have an Account that stores a List of addresses. These addresses are simply Strings. We define the Meta-Data like this

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastName" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="java.lang.String"/>
            <join/>
            <element column="ADDRESS"/>
        </field>
    </class>
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String. In addition we have an additional index column to form part of the primary key (along with the FK back to the ACCOUNT table). You can override the default naming of this column by specifying the <order> tag.

Embedded into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.

Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the serialised-element attribute on the collection MetaData element. This is described in Serialised Collection Elements

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.4.4 1-to-N (Map)

JDO 1-N/N-1 Relationships with Maps

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Map of objects of another class. There are two general ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Map.

The various possible relationships are described below.

- Map[PC, PC] using join table
- Map[Simple, PC] using join table
- Map[PC, Simple] using join table
- Map[Simple, Simple] using join table
- 1-N Bidirectional using Foreign-Key (key stored in the value class)
- 1-N Unidirectional using Foreign-Key (key stored in the value class)
- 1-N Unidirectional using Foreign-Key (value stored in the key class)
- 1-N embedded keys/values using Join Table
- 1-N Serialised map
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in value)

This page is aimed at Map fields and so applies to fields of Java type *java.util.HashMap, java.util.Hashtable, java.util.LinkedHashMap, java.util.Map, java.util.SortedMap, java.util.TreeMap, java.util.Properties*

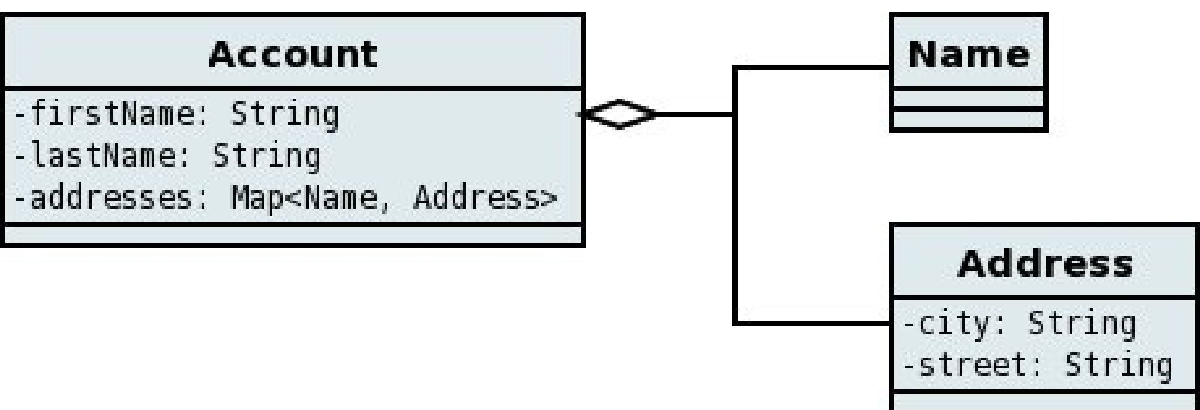
1-N Map using Join Table

We have a class Account that contains a Map. With a Map we store values using keys. As a result we have 3 main combinations of key and value, bearing in mind whether the key or value is PersistenceCapable.

Map[PC, PC]

Here both the keys and the values are PersistenceCapable. Like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <map key-type="com.mydomain.Name" value-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>

    <class name="Name" identity-type="datastore">
    </class>
</package>
```

This will create 4 tables in the datastore, one for Account, one for Address, one for Name and a join table containing foreign keys to the key/value tables.

If you want to configure the names of the columns in the "join" table you would use the <key> and <value> subelements of <field>, something like this

```
        <field name="addresses" persistence-modifier="persistent"
table="ACCOUNT_ADDRESS">
            <map key-type="com.mydomain.Name" value-type="com.mydomain.Address"/>
            <join>
                <column name="ACCOUNT_ID"/>
            </join>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <key>
                <column name="NAME_ID"/>
            </key>
            <value>
                <column name="ADDRESS_ID"/>
            </value>
        </field>
```

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and key table, specify <foreign-key> below the <key> element.
- To specify the foreign-key between join table and value table, specify <foreign-key> below the <value> element.

Which changes the names of the join table to ACCOUNT_ADDRESS from ACCOUNT_ADDRESSES and the names of the columns in the join table from ACCOUNT_ID_OID to ACCOUNT_ID, from NAME_ID_KID to NAME_ID, and from ADDRESS_ID_VID to ADDRESS_ID.

Map[Simple, PC]

Here our key is a simple type (in this case a String) and the values are PersistenceCapable. Like this

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <join/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

This will create 3 tables in the datastore, one for Account, one for Address and a join table also containing the key.

If you want to configure the names of the columns in the "join" table you would use the <key> and <value> subelements of <field> as shown above.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

Map[PC, Simple]

This operates exactly the same as "Map[Simple, PC]" except that the additional table is for the key instead of the value.

Map[Simple, Simple]

Here our keys and values are of simple types (in this case a String). Like this

Account
-firstName: String -lastName: String -addresses: Map<String, String>

If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <map key-type="java.lang.String" value-type="java.lang.String"/>
            <join/>
        </field>
    </class>
</package>
```

This results in just 2 tables. The "join" table contains both the key AND the value.

If you want to configure the names of the columns in the "join" table you would use the <key> and <value> subelements of <field> as shown above.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

Embedded

The above relationship types assume that all PersistenceCapable classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the keys, the values, or the keys and the values of the map into this join table. This is described in Embedded Maps.

1-N Map using Foreign-Key

1-N Foreign-Key Bidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value.

With these classes we want to store a foreign-key in the value table (ADDRESS), and we want to use the "alias" field in the Address class as the key to the map. If you define the Meta-Data for these classes as follows

```

<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
            <key mapped-by="alias"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="account" persistence-modifier="persistent">
        </field>
        <field name="alias" null-value="exception">
            <column name="KEY" length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

This will create 2 tables in the datastore. One for Account, and one for Address. The table for Address will contain the key field as well as an index to the Account record (notated by the mapped-by tag).

1-N Foreign-Key Unidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value. As in the case of the bidirectional relation above we're using a field (*alias*) in the Address class as the key of the map.

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
            <key mapped-by="alias"/>
            <value column="ACCOUNT_ID_OID"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="alias" null-value="exception">
            <column name="KEY" length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified on the "field" element, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the value element within the field of the map.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then you should use the "Join Table" relationship above.

1-N Foreign-Key Unidirectional (value stored in key)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the key. We're using a field (*businessAddress*) in the Address class as the value of the map.

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. We don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="phoneNumbers" persistence-modifier="persistent">
            <map key-type="com.mydomain.Address" value-type="java.lang.String"/>
            <key column="ACCOUNT_ID_OID"/>
            <value mapped-by="businessPhoneNumber"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="businessPhoneNumber" null-value="exception">
            <column name="BUS_PHONE" length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

There will be 2 tables, one for Address, and one for Account. The key thing here is that we have specified a "mapped-by" on the "value" element. Note that we have no "mapped-by" attribute specified on the "field" element, and also no "join" element. If you wish to specify the names of the columns used in the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

schema for the foreign key in the Address table you should use the key element within the field of the map.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then you should use the "Join Table" relationship above.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.5 N-to-1

JDO N-1 Relationships

You have a N-to-1 relationship when an object of a class has an associated object of another class (only one associated object) and several of this type of object can be linked to the same associated object. From the other end of the relationship it is effectively a 1-N, but from the point of view of the object in question, it is N-1. You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only the "N" side knows about the other class (unidirectional). These are described below.

Unidirectional

For this case you could have 2 classes, User and Account, as below.
so the Account class ("N" side) knows about the User class ("1" side), but not vice-versa. A particular user could be related to several accounts. If you define the Meta-Data for these classes as follows

```
<package name="mydomain">
    <class name="User" identity-type="datastore">
        <field name="login" persistence-modifier="persistent">
            <column length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>

    <class name="Account" identity-type="datastore">
        <field name="firstName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="secondName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="user" persistence-modifier="persistent">
        </field>
    </class>
</package>
```

This will create 2 tables in the database, one for User (with name USER), and one for Account (with name ACCOUNT and a column USER_ID), as shown below.

Things to note :-

- If you wish to specify the names of the database tables and columns for these classes, you can use the attribute table (on the class element) and the attribute name (on the column element)
- If you call PM.deletePersistent() on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Bidirectional

This relationship is described in the guide for 1-N relationships. In particular there are 2 ways to define the relationship. The first uses a Join Table to hold the relationship. The second uses a Foreign Key in the "N" object to hold the relationship. Please refer to the 1-N relationships bidirectional relations since they show this exact relationship.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.6 M-to-N

JDO M-N Relationships

You have a M-to-N (or Many-to-Many) relationship if an object of a class A has associated objects of class B, and class B has associated objects of class A. This relationship may be achieved through Java Set, Map, List or subclasses of these, although the only one that supports a true M-N is for a Set/Collection.

With DataNucleus this can be set up as described in this section, using what is called a Join Table relationship. Let's take the following example and describe how to model it with the different types of collection classes. We have 2 classes, Product and Supplier as below.

Here the Product class knows about the Supplier class. In addition the Supplier knows about the Product class, however with DataNucleus (as with the majority of JDO implementations) these relationships are independent.

Using Set

If you define the Meta-Data for these classes as follows

```
<package name="mydomain">
    <class name="Product" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="price" persistence-modifier="persistent">
        </field>
        <field name="suppliers" persistence-modifier="persistent"
table="PRODUCTS_SUPPLIERS">
            <collection element-type="mydomain.Supplier"/>
            <join>
                <column name="PRODUCT_ID"/>
            </join>
            <element>
                <column name="SUPPLIER_ID"/>
            </element>
        </field>
    </class>

    <class name="Supplier" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="products" persistence-modifier="persistent"
mapped-by="suppliers">
            <collection element-type="mydomain.Product"/>
        </field>
    </class>
</package>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Note how we have specified the information only once regarding join table name, and join column names as well as the <join>. This is the JDO standard way of specification, and results in a single join table.

See also :-

- M-N Worked Example
- M-N with Attributes Worked Example

Using List

Firstly a true M-N relation with Lists is impossible since there are two lists, and it is undefined as to which one applies to which side etc. What is shown below is two independent 1-N unidirectional join table relations. If you define the Meta-Data for these classes as follows

```
<package name="mydomain">
    <class name="Product" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="price" persistence-modifier="persistent"/>
        <field name="suppliers" persistence-modifier="persistent">
            <collection element-type="mydomain.Supplier"/>
            <join/>
        </field>
    </class>

    <class name="Supplier" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="products" persistence-modifier="persistent">
            <collection element-type="mydomain.Product"/>
            <join/>
        </field>
    </class>
</package>
```

There will be 4 tables, one for Product, one for Supplier, and the join tables. The difference from the Set example is in the contents of the join tables. An index column is added to keep track of the position of objects in the Lists.

In the case of a List at both ends it doesn't make sense to use a single join table because the ordering can only be defined at one side, so you have to have 2 join tables.

Using Map

If you define the Meta-Data for these classes as follows

```
<package name="mydomain">
    <class name="Product" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="price" persistence-modifier="persistent">
        </field>
        <field name="suppliers" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="mydomain.Supplier"/>
            <join/>
        </field>
    </class>

    <class name="Supplier" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="products" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="mydomain.Product"/>
            <join/>
        </field>
    </class>
</package>
```

This will create 4 tables in the datastore, one for Product, one for Supplier, and the join tables which also contains the keys to the Maps (a String).

Relationship Behaviour

Please be aware of the following.

- To add an object to an M-N relationship you need to set it at both ends of the relation since the relation is bidirectional and without such information the JDO implementation won't know which end of the relation is correct.
- If you want to delete an object from one end of a M-N relationship you will have to remove it first from the other objects relationship. If you don't you will get an error message that the object to be deleted has links to other objects and so cannot be deleted.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.7 Compound Identity Relation

JDO Compound Identity Relationships

An identifying relationship (or "compound identity relationship" in JDO) is a relationship between two objects of two classes in which the child object must coexist with the parent object and where the primary key of the child includes the PersistenceCapable object of the parent. So effectively the key aspect of this type of relationship is that the primary key of one of the classes includes a PersistenceCapable field (hence why is is referred to as Compound Identity). This type of relation is available in the following forms

- 1-1 unidirectional
- 1-N collection bidirectional using ForeignKey
- 1-N map bidirectional using ForeignKey (key stored in value)

1-1 Relationship

Lets take the same classes as we have in the 1-1 Relationships. In the 1-1 relationships guide we note that in the datastore representation of the **User** and **Account** the ACCOUNT table has a primary key as well as a foreign-key to USER. In our example here we want to just have a primary key that is also a foreign-key to USER. To do this we need to modify the classes slightly and add primary-key fields and use "application-identity".

In addition we need to define primary key classes for our User and Account classes

```
public class User
{
    long id;

    ... (remainder of User class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Account
{
    User user;

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public User.PK user; // Use same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");

            this.user = new User.PK(token.nextToken());
        }

        public String toString()
        {
            return "" + this.user.toString();
        }

        public int hashCode()
        {
            return user.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.user.equals(otherPK.user);
            }
            return false;
        }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<package name="mydomain">
    <class name="User" identity-type="application" objectid-class="User$PK">
        <field name="id" primary-key="true"/>
        <field name="login" persistence-modifier="persistent">
            <column length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>

    <class name="Account" identity-type="application" objectid-class="Account$PK">
        <field name="user" persistence-modifier="persistent" primary-key="true">
            <column name="USER_ID"/>
        </field>
        <field name="firstName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="secondName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

So now we have the following datastore schema

Things to note :-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- You can only have one "Account" object linked to a particular "User" object since the FK to the "User" is now the primary key of "Account". To remove this restriction you could also add a "long id" to "Account" and make the "Account.PK" a composite primary-key

1-N Collection Relationship

Lets take the same classes as we have in the 1-N Relationships (FK). In the 1-N relationships guide we note that in the datastore representation of the **Account** and **Address** classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to *include* the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class Account
{
    long id; // PK field

    Set addresses = new HashSet();

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Address
{
    long id;
    Account account;

    .. (remainder of Address class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id; // Same name as real field above
        public Account.PK account; // Same name as the real field above

        public PK()
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");
            this.id = Long.valueOf(token.nextToken()).longValue();
            this.account = new Account.PK(token.nextToken());
        }

        public String toString()
        {
            return "" + id + "::" + this.account.toString();
        }

        public int hashCode()
        {
            return (int)id ^ account.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id &&
this.account.equals(otherPK.account);
            }
            return false;
        }
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<package name="mydomain">
    <class name="Account" identity-type="application" objectid-class="Account$PK">
        <field name="id" primary-key="true"/>
        <field name="firstName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="secondName" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <collection element-type="Address"/>
        </field>
    </class>

    <class name="Address" identity-type="application" objectid-class="Address$PK">
        <field name="id" primary-key="true"/>
        <field name="account" persistence-modifier="persistent" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

So now we have the following datastore schema

Things to note :-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- If we had omitted the "id" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "id" field too.

1-N Map Relationship

Lets take the same classes as we have in the 1-N Relationships (FK). In this guide we note that in the datastore representation of the **Account** and **Address** classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to *include* the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes

```
public class Account
{
    long id; // PK field

    Set addresses = new HashSet();

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Address
{
    String alias;
    Account account;

    .. (remainder of Address class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public String alias; // Same name as real field above
        public Account.PK account; // Same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");
            this.alias = Long.valueOf(token.nextToken()).longValue();
            this.account = new Account.PK(token.nextToken());
        }

        public String toString()
        {
            return alias + "::" + this.account.toString();
        }

        public int hashCode()
        {
            return alias.hashCode() ^ account.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            {
                PK otherPK = (PK)other;
                return otherPK.alias.equals(this.alias) &&
this.account.equals(otherPK.account);
            }
            return false;
        }
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<package name="com.mydomain">
    <class name="Account" objectid-class="Account$PK">
        <field name="id" primary-key="true"/>
        <field name="firstname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="lastname" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="addresses" persistence-modifier="persistent"
mapped-by="account">
            <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
            <key mapped-by="alias"/>
        </field>
    </class>

    <class name="Address" objectid-class="Address$PK>
        <field name="account" persistence-modifier="persistent" primary-key="true"/>
        <field name="alias" null-value="exception" primary-key="true">
            <column name="KEY" length="20" jdbc-type="VARCHAR"/>
        </field>
        <field name="city" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
        <field name="street" persistence-modifier="persistent">
            <column length="50" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>
```

So now we have the following datastore schema

Things to note :-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- If we had omitted the "alias" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

the "alias" field too as part of the PK.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.1 JPA Class Mapping

JPA Class Mapping

When persisting a class you need to decide how it is to be mapped to the datastore. By this we mean which fields of the class are persisted. DataNucleus knows how to persist certain Java types and so you bear this list in mind when deciding which fields to persist. Also please note that JPA **cannot persist static or final fields**. Let's take a sample class as an example

```
public class Hotel
{
    private long id; // identity
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    private String hotelNickname;
    private Set rooms = new HashSet();
    private Manager manager;
    ...
}
```

We have a series of fields and we want to persist all fields apart from *hotelNickname* which is of no real use in our system. In addition, we want our *Hotel* class to be detachable, meaning that we can detach objects of that type update them in a different part of our system, and the attach them again.

We can define this basic persistence information in 3 ways - with XML MetaData, with JDK1.5 Annotations or with a mix of MetaData and Annotations. We show all ways here.

MetaData

To achieve the above aim we define our Meta-Data like this

```
<entity class="org.datanucleus.test.Hotel">
    <attributes>
        <id name="id"/>
        <basic name="name"/>
        <basic name="address"/>
        <basic name="telephoneNumber"/>
        <basic name="numberOfRooms"/>
        <transient name="hotelNickname"/>
        <one-to-many name="rooms" target-entity="org.datanucleus.test.Room"/>
        <one-to-one name="manager" target-entity="org.datanucleus.test.Manager"/>
    </attributes>
</entity>
```

Note the following

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- We have identified our *id* field as the identity. We didnt bother specifying an Primary key class since we only have a single PK field
- We have included all fields in the MetaData, although if you look at the Types Guide you see the column "Persistent?". All "simple" types like String, int are by default persistent and so we could have omitted these since they are by default going to be persisted.
- We have used *transient* for the *hotelNickname* field to make it non-persistent.
- Our Set field we have identified as "1-N" and the type of the element that it contains. This is compulsory for collection and map fields (unless using JDK1.5 generics).
- We have identified our *manager* field as "1-1"

So it is really very simple. This first step is to define the basic persistence of a class. If you are using a datastore (such as RDBMS) that requires detailed mapping information then you now need to proceed to the JPA Schema Mapping Guide. If however you are using a datastore that doesn't need such information (such as DB4O) then you have defined the persistence of your class.

See also :-

- JPA MetaData reference

Annotations

Here we are using JDK1.5 or higher and we annotate the class directly using JPA Annotations We annotate the class like this

```
@Entity
public class Hotel
{
    @Id
    private long id;

    @Basic
    private String name;

    @Basic
    private String address;

    @Basic
    private String telephoneNumber;

    @Basic
    private int numberOfRooms;

    @Transient
    private String hotelNickname;

    @OneToMany(target="Room")
    private Set rooms = new HashSet();

    @OneToOne(target="Manager)
    private Manager manager;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

See also :-

- JPA Annotations reference

Annotations + MetaData

If we are using JDK 1.5+ we can take advantage of Annotations, but we want to take into account the disadvantage of Annotations, namely that we may want to deploy our application to multiple datastores. This means that we would be extremely unwise to specify ORM information in Annotations. With this in mind we decide that we will specify just the basic persistence information (which classes/fields are persisted etc) using Annotations, and the remainder will go in MetaData.

The order of precedence for persistence information is

- JPA MetaData definition
- Annotations definition

So anything specified in MetaData will override all Annotations.

Persistence Aware

With JPA you cannot access public fields of classes. DataNucleus allows an extension to permit this, but such classes need special enhancement. To allow this you need to

- Annotate the class that will access these public fields (assuming it isn't an Entity) with the DataNucleus extension annotation @PersistenceAware
- At runtime set the persistence property **datanucleus.jpa.level** to *DataNucleus*

You perform the annotation of the class as follows

```
@PersistenceAware
public class MyClassThatAccessesPublicFields
{
    ...
}
```

See also :-

- Annotations reference for @PersistenceAware

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.2 Java Types

JPA : Persistable Java Types

When persisting a class, a persistence solution needs to know how to persist the types of each field in the class. Clearly a persistence solution can only support a finite number of Java types; it cannot know how to persist every possible type creatable. The JPA specification define lists of types that are required to be supported by all implementations of those specifications. This support can be conveniently split into two parts

First-Class (FCO) Types

An object that can be *referred to* (object reference, providing a relation) and that has an "identity" is termed a **First Class Object (FCO)**. DataNucleus supports the following Java types as FCO

- **persistable** : any class marked for persistence (annotations or XML) can be persisted with its own identity in the datastore
- **interface** where the field represents a *persistable* object
- **java.lang.Object** where the field represents a *persistable* object

Supported Second-Class (SCO) Types

An object that does not have an "identity" is termed a **Second Class Object (SCO)**. This is something like a String or Date field in a class, or alternatively a Collection (that contains other objects). The table below shows the currently supported SCO java types in DataNucleus. The table shows

- **Extension?** : whether the type is JPA standard, or is a DataNucleus extension
- **default-fetch-group (DFG)** : whether the field is retrieved by default when retrieving the object itself
- **persistence-modifier** : whether the field is persisted by default, or whether the user has to mark the field as persistent in XML/annotations to persist it
- **proxied** : whether the field is represented by a "proxy" that intercepts any operations to detect whether it has changed internally.
- **primary-key** : whether the field can be used as part of the primary-key

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
boolean		✓	✓	✗	✓
byte		✓	✓	✗	✓
char		✓	✓	✗	✓
double		✓	✓	✗	✗

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
float		✓	✓	✗	✗
int		✓	✓	✗	✓
long		✓	✓	✗	✓
short		✓	✓	✗	✓
boolean[]	Extension	✗	✓	✗	✗
byte[]		✗	✓	✗	✗
char[]		✗	✓	✗	✗
double[]	Extension	✗	✓	✗	✗
float[]	Extension	✗	✓	✗	✗
int[]	Extension	✗	✓	✗	✗
long[]	Extension	✗	✓	✗	✗
short[]	Extension	✗	✓	✗	✗
java.lang.Boolean		✓	✓	✗	✓
java.lang.Byte		✓	✓	✗	✓
java.lang.Character		✓	✓	✗	✓
java.lang.Double		✓	✓	✗	✗
java.lang.Float		✓	✓	✗	✗
java.lang.Integer		✓	✓	✗	✓
java.lang.Long		✓	✓	✗	✓
java.lang.Short		✓	✓	✗	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.lang.Boolean[]	Extension	✗	✓	✗	✗
java.lang.Byte[]		✗	✓	✗	✗
java.lang.Character[]		✗	✓	✗	✗
java.lang.Double[]	Extension	✗	✓	✗	✗
java.lang.Float[]	Extension	✗	✓	✗	✗
java.lang.Integer[]	Extension	✗	✓	✗	✗
java.lang.Long[]	Extension	✗	✓	✗	✗
java.lang.Short[]	Extension	✗	✓	✗	✗
java.lang.Number [4]	Extension	✓	✓	✗	✗
java.lang.Object	Extension	✗	✗	✗	✗
java.lang.String		✓	✓	✗	✓
java.lang.StringBuffer [3]	Extension	✗	✓	✗	✓
java.lang.String[]	Extension	✗	✓	✗	✗
java.math.BigDecimal		✓	✓	✗	✗
java.math.BigInteger		✓	✓	✗	✓
java.math.BigDecimal[]	Extension	✗	✓	✗	✗
java.math.BigInteger[]	Extension	✗	✓	✗	✗
java.sql.Date		✗	✗	✓	✓
java.sql.Time		✗	✗	✓	✓
java.sql.Timestamp		✗	✗	✓	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.util.ArrayList		✗	✓	✓	✗
java.util.BitSet		✗	✗	✓	✗
java.util.Calendar		✗	✗	✓	✗
java.util.Collection		✗	✓	✓	✗
java.util.Currency	Extension	✗	✓	✗	✓
java.util.Date		✓	✓	✓	✓
java.util.Date[]	Extension	✗	✓	✗	✗
java.util.GregorianCalendar [7]		✗	✗	✓	✗
java.util.HashMap		✗	✓	✓	✗
java.util.HashSet		✗	✓	✓	✗
java.util.Hashtable		✗	✓	✓	✗
java.util.LinkedHashMap [5]		✗	✓	✓	✗
java.util.LinkedHashSet [6]		✗	✓	✓	✗
java.util.LinkedList		✗	✓	✓	✗
java.util.List		✗	✓	✓	✗
java.util.Locale	Extension	✗	✓	✗	✓
java.util.Locale[]	Extension	✗	✓	✗	✗
java.util.Map		✗	✓	✓	✗
java.util.Properties	Extension	✗	✗	✓	✗
java.util.PriorityQueue	Extension	✗	✗	✓	✗

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Extension?	DFG?	Persistent?	Proxied?	PK?
java.util.Queue	Extension	✗	✗	✓	✗
java.util.Set		✗	✓	✓	✗
java.util.SortedMap [2]	Extension	✗	✗	✓	✗
java.util.SortedSet [1]	Extension	✗	✗	✓	✗
java.util.Stack	Extension	✗	✗	✓	✗
java.util.TimeZone		✗	✗	✗	✓
java.util.TreeMap [2]		✗	✓	✓	✗
java.util.TreeSet [1]		✗	✓	✓	✗
java.util.UUID	Extension	✗	✗	✗	✓
java.util.Vector		✗	✓	✗	✗
java.awt.Color	Extension	✗	✗	✗	✗
java.awt.Point	Extension	✗	✗	✓	✗
java.awt.image.Buffered	Extension	✗	✗	✗	✗
java.net.URI	Extension	✗	✗	✗	✓
java.net.URL	Extension	✗	✗	✗	✓
java.io.Serializable		✗	✗	✗	✗
persistable		✗	✗	✗	✓
persistable[]	Extension	✗	✗	✗	✗
java.lang.Enum		✓	✓	✗	✓
java.lang.Enum[]		✗	✓	✗	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- [1] - java.util.SortedSet, java.util.TreeSet allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <collection>). The headSet, tailSet, subSet methods are only supported when using cached collections.
- [2] - java.util.SortedMap, java.util.TreeMap allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <map>). The headMap, tailMap, subMap methods are only supported when using cached containers.
- [3] - java.lang.StringBuffer dirty check mechanism is limited to immutable mode, it means, if you change a StringBuffer object field, you must reassign it to the owner object field to make sure changes are propagated to the database.
- [4] - java.lang.Number will be stored in a column capable of storing a BigDecimal, and will store to the precision of the object to be persisted. On reading back the object will be returned typically as a BigDecimal since there is no mechanism for determing the type of the object that was stored.
- [5] - java.util.LinkedHashMap treated as a Map currently. No List-ordering is supported.
- [6] - java.util.LinkedHashSet treated as a Set currently. No List-ordering is supported.
- [7] - java.util.Calendar can be stored into two columns (millisecs, Timezone) or into a single column (Timestamp). The single column option is not guaranteed to preserve the TimeZone of the input Calendar.

Note that support is available for persisting other types depending on the datastore to which you are persisting

- RDBMS GeoSpatial types via the DataNucleus RDBMS Spatial plugin

If you have support for any additional types and would either like to contribute them, or have them listed here, let us know

Plugin DataNucleus allows you the luxury of being able to provide SCO support for your own Java types when using RDBMS datastores

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.3 Application Identity

Application Identity

With application identity you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key class (unless using SingleFieldIdentity, where one is provided for you), and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. With application identity the field(s) of the primary key will be present as field(s) of the class itself. To specify that a class is to use application identity, you add the following to the MetaData for the class.

```
<entity class="org.mydomain.MyClass">
    <id-class class="org.mydomain.MyIdClass"/>
    <attributes>
        <id name="myPrimaryKeyField"/>
    </attributes>
</entity>
```

For JPA1 we specify the id field and id-class. Alternatively, if we are using annotations

```
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass
{
    @Id
    private long myPrimaryKeyField;
}
```

When you have an inheritance hierarchy, you should specify the identity type and any primary-key fields in the base class for the inheritance tree. This is then used for all persistent classes in the tree.

See also :-

- MetaData reference for <id> element
- Annotations reference for @Id

Primary Key

Using application identity requires the use of a Primary Key class.With JPA when you have a single-field you don't need to provide a primary key class. Where the class has multiple fields that form the primary key a Primary Key class must be provided. In JPA1 when there is a single primary key field you dont need to specify the primary key class. If there are more than 1 "id" fields then you define the id-class.

See also :-

- Primary Key Guide - user-defined and built-in primary keys

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Generating identities

By choosing application identity you are controlling the process of identity generation for this class. This does not mean that you have a lot of work to do for this. JPA1 defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

See also :-

- Identity Generation Guide - strategies for generating ids

Changing Identities

JPA doesn't define what happens if you change the identity (an identity field) of an object once persistent. DataNucleus doesn't currently support changes to identities.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.4 Primary Keys

Primary Key Classes

As has been described in the application identity guide, when you choose application identity you are defining which fields of the class are part of the primary key, and you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key (PK) class, and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. You specify the primary key class like this

```
<entity class="MyClass">
    <id-class class="MyIdClass"/>
    ...
</entity>
```

or using annotations

```
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass
{
    ...
}
```

You now need to define the PK class to use. This is simplified for you because **if you have only one PK field then you dont need to define a PK class** and you only define it when you have a composite PK.

An important thing to note is that the PK can only be made up of fields of the following Java types

- Primitives : **boolean**, **byte**, **char**, **int**, **long**, **short**
- java.lang : **Boolean**, **Byte**, **Character**, **Integer**, **Long**, **Short**, **String**, **Enum**, StringBuffer
- java.math : **BigInteger**
- java.sql : **Date**, **Time**, **Timestamp**
- java.util : **Date**, Currency, Locale, TimeZone, UUID
- java.net : URI, URL
- javax.jdo.spi : **PersistenceCapable**

Note that the types in **bold** are JPA standard types. Any others are DataNucleus extensions and, as always, check the specific datastore docs to see what is supported for your datastore.

Single primary-key field

The simplest way of using application identity is where you have a single PK field, and in this case you

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

use an inbuilt primary key class that DataNucleus provides, so you don't need to specify the objectid-class. Let's take an example

```
public class MyClass
{
    long id;
    String name;
    String description;
    ...
}

<entity class="MyClass">
    <attributes>
        <id name="id"/>
        <basic name="name"/>
        <basic name="description"/>
    </attributes>
</entity>
```

So we didnt specify the JPA "id-class". You will, of course, have to give the field a value before persisting the object, either by setting it yourself, or by using a value-strategy on that field.

Rules for User-Defined Primary Key classes

If you wish to use application identity and don't want to use the "SingleFieldIdentity" builtin PK classes then you must define a Primary Key class of your own. You can't use classes like java.lang.String, or java.lang.Long directly. You must follow these rules when defining your primary key class.

- the Primary Key class must be public
- the Primary Key class must implement Serializable
- the Primary Key class must have a public no-arg constructor, which might be the default constructor
- the field types of all non-static fields in the Primary Key class must be serializable, and are recommended to be primitive, String, Date, or Number types
- all serializable non-static fields in the Primary Key class must be public
- the names of the non-static fields in the Primary Key class must include the names of the primary key fields in the JDO class, and the types of the common fields must be identical
- the equals() and hashCode() methods of the Primary Key class must use the value(s) of all the fields corresponding to the primary key fields in the JDO class
- if the Primary Key class is an inner class, it must be static
- the Primary Key class must override the toString() method defined in Object, and return a String that can be used as the parameter of a constructor
- the Primary Key class must provide a String constructor that returns an instance that compares equal to an instance that returned that String by the toString() method.
- the Primary Key class must be only used within a single inheritence tree.

Primary Key Example - Multiple Field

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here's an example of a composite (multiple field) primary key class

```
public class ComposedIdKey implements Serializable
{
    public String field1;
    public String field2;

    /**
     *  Default constructor.
     */
    public ComposedIdKey ()
    {
    }

    /**
     * String constructor.
     */
    public ComposedIdKey(String value)
    {
        StringTokenizer token = new StringTokenizer (value, "::");
        //className
        token.nextToken ();
        //field1
        this.field1 = token.nextToken ();
        //field2
        this.field2 = token.nextToken ();
    }

    /**
     * Implementation of equals method.
     */
    public boolean equals(Object obj)
    {
        if (obj == this)
        {
            return true;
        }
        if (!(obj instanceof ComposedIdKey))
        {
            return false;
        }
        ComposedIdKey c = (ComposedIdKey)obj;

        return field1.equals(c.field1) && field2.equals(c.field2);
    }

    /**
     *  Implementation of hashCode method that supports the
     *  equals-hashCode contract.
     */
    public int hashCode ()
    {
        return this.field1.hashCode() ^ this.field2.hashCode();
    }

    /**
     *  Implementation of toString that outputs this object id's PK values.
     */
    public String toString ()
    {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        return this.getClass().getName() + "::"  + this.field1 + "::" + this.field2;
    }
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.5 Fields/Properties

Persistent Fields or Properties

There are two distinct modes of persistence definition. The most common uses **fields**, whereas an alternative uses **properties**.

Persistent Fields

The most common form of persistence is where you have a **field** in a class and want to persist it to the datastore. With this mode of operation DataNucleus will persist the values stored in the fields into the datastore, and will set the values of the fields when extracting it from the datastore.

Requirement : you have a field in the class. This can be public, protected, private or package access, but cannot be static or final.

An example of how to define the persistence of a field is shown below

```
@Entity
public class MyClass
{
    @Basic
    Date birthday;
}
```

So, using annotations, we have marked this class as persistent, and the field also as persistent. Using XML MetaData we would have done

```
<entity name="mydomain.MyClass">
    <attributes>
        <basic name="birthday"/>
    </attributes>
</entity>
```

Persistent Properties

A second mode of operation is where you have Java Bean-style getter/setter for a **property**. In this situation you want to persist the output from *getXXX* to the datastore, and use the *setXXX* to load up the value into the object when extracting it from the datastore.

Requirement : you have a property in the class with Java Bean getter/setter methods. These methods can be public, protected, private or package access, but cannot be static. The class must have BOTH getter AND setter methods.

An example of how to define the persistence of a property is shown below

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Entity
public class MyClass
{
    @Basic
    Date getBirthday()
    {
        ...
    }

    void setBirthday(Date date)
    {
        ...
    }
}
```

So, using annotations, we have marked this class as persistent, and the getter is marked as persistent. Using XML MetaData we would have done

```
JPA:
<entity name="mydomain.MyClass">
    <attributes>
        <basic name="birthday"/>
    </attributes>
</entity>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.6 Value Generation

Value generation

Fields of a class can either have the values set by you the user, or you can set DataNucleus to generate them for you. This is of particular importance with identity fields where you want unique identities. You can use this value generation process with the identity field(s) in JPA. There are many different "strategies" for generating values, as defined by the JPA specification. Some strategies are specific to a particular datastore, and some are generic. You should choose the strategy that best suits your target datastore. The available strategies are :-

- auto - this is the default and allows DataNucleus to choose the most suitable for the datastore
- sequence - this uses a datastore sequence (if supported by the datastore)
- identity - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- table - this is datastore neutral and increments a sequence value using a table.

See also :-

- JPA MetaData reference for <generated-value>
- JPA Annotation reference for @GeneratedValue

auto

With this strategy DataNucleus will choose the most appropriate strategy for the datastore being used. If you also specify the 'sequence' name attribute and the datastore supports sequences then "sequence" strategy would be used. Otherwise it will always choose "increment" strategy.

sequence

A sequence is a user-defined database function that generates a sequence of unique numeric ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. DataNucleus supports sequences for the following datastores:

- Oracle
- PostgreSQL
- SAP DB
- DB2
- Firebird
- DB4O

To configure a class to use either of these generation methods using application identity you would add the following to the class' Meta-Data

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<entity class="MyClass">
    <attributes>
        <id name="myId">
            <generated-value strategy="sequence"/>
        </id>
    </attributes>
</entity>
```

or using annotations

```
@Entity
public class MyClass
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    private long myfield;
    ...
}
```

If the sequence does not yet exist in the database at the time DataNucleus needs a new unique identifier, a new sequence is created in the database based on the JPA Meta-Data configuration. Additional properties for configuring sequences are set in the JPA Meta-Data, see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

Property	Description	Required
key-cache-size	number of unique identifiers to cache in the PersistenceManagerFactory instance. Notes: 1. This setting SHOULD match the *key-start-with* setting value if *key-start-with* is provided, otherwise it can cause duplicate keys errors when inserting new objects into the database. 2. The keys are pre-allocated, cached and used on demand. If *key-cache-size* is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.	No. Defaults to 1.
key-min-value	determines the minimum value a sequence can generate	No
key-max-value	determines the maximum value a sequence can generate	No
key-start-with	the initial value for the sequence	No
key-increment-by	specifies which value is added to the current sequence value to create a new value. default is 1	No
key-database-cache-size	specifies how many sequence numbers are to be preallocated and stored in memory for faster access. This is an optimization feature provided by the database	No
sequence-catalog-name	Name of the catalog where the sequence is.	No.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Property	Description	Required
sequence-schema-name	Name of the schema where the sequence is.	No.

This value generator will generate values unique across different JVMs

identity

Auto-increment/identity/serial are primary key columns that are populated when a row is inserted in the table. These use the databases own keywords on table creation and so rely on having the table structure either created by DataNucleus or having the column with the necessary keyword.

DataNucleus supports auto-increment/identity/serial keys for many databases including :

- DB2 (IDENTITY)
- MySQL (AUTOINCREMENT)
- MSSQL (IDENTITY)
- Sybase (IDENTITY)
- HSQLDB (IDENTITY)
- H2 (IDENTITY)
- PostgreSQL (SERIAL)

This generation strategy should only be used if there is a single "root" table for the inheritance tree. If you have more than 1 root table (e.g using subclass-table inheritance) then you should choose a different generation strategy

For a class using application identity you need to set the *value-strategy* attribute on the primary key field. You can configure the Meta-Data for the class something like this

```
<entity class="MyClass">
    <attributes>
        <id name="myId">
            <generated-value strategy="identity"/>
        </id>
    </attributes>
</entity>
```

Please be aware that if you have an inheritance tree with the base class defined as using "identity" then the column definition for the PK of the base table will be defined as "AUTO_INCREMENT" or "IDENTITY" or "SERIAL" (dependent on the RDBMS) and all subtables will NOT have this identifier added to their PK column definitions. This is because the identities are assigned in the base table (since all objects will have an entry in the base table).

This value generator will generate values unique across different JVMs

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

table

This method is database neutral and uses a sequence table that holds an incrementing sequence value. The unique identifier value returned from the database is translated to a java type: java.lang.Long. This strategy will work with any datastore. This method require a sequence table in the database and creates one if doesn't exist.

To configure an application identity class to use this generation method you simply add this to the class' Meta-Data. If your class is in an inheritance tree you should define this for the base class only.

```
<entity class="MyClass">
    <attributes>
        <id name="myId">
            <generated-value strategy="table"/>
        </id>
    </attributes>
</entity>
```

Additional properties for configuring this generator are set in the JPA Meta-Data, see the available properties below. Unsupported properties are silently ignored by DataNucleus.

Property	Description	Required
key-initial-value	First value to be allocated.	No. Defaults to 1
key-cache-size	number of unique identifiers to cache. The keys are pre-allocated, cached and used on demand. If *key-cache-size* is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.	No. Defaults to 5
sequence-table-basis	Whether to define uniqueness on the base class name or the base table name. Since there is no "base table name" when the root class has "subclass-table" this should be set to "class" when the root class has "subclass-table" inheritance	No. Defaults to *class*, but the other option is table
sequence-name	name for the sequence (overriding the "sequence-table-basis" above). The row in the table will use this in the PK column	No
sequence-table-name	Table name for storing the sequence.	No. Defaults to *SEQUENCE_TABLE*
sequence-catalog-name	Name of the catalog where the table is.	No.
sequence-schema-name	Name of the schema where the table is.	No.
sequence-name-column-name	Name for the column that represent sequence names.	No. Defaults to *SEQUENCE_NAME*
sequence-nextval-column-name	Name for the column that represent incremeting sequence values.	No. Defaults to *NEXT_VAL*

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Property	Description	Required
table-name	Name of the table whose column we are generating the value for (used when we have no previous sequence value and want a start point.	No.
column-name	Name of the column we are generating the value for (used when we have no previous sequence value and want a start point.	No.

This value generator will generate values unique across different JVMs

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.7 JPA MetaData

JPA Metadata Overview

JPA requires the persistence of classes to be defined via Metadata. This Metadata can be provided in the following forms

- XML : the traditional mechanism, with XML files containing information for each class to be persisted.
- Annotations : using JDK1.5+ annotations in the classes to be persisted

Metadata priority

JPA defines the priority order for metadata as being

- JPA XML Metadata
- Annotations

If a class has annotations and JPA XML Metadata then the XML Metadata will take precedence over the annotations (or rather be merged on top of the annotations).

XML Metadata validation

By default any XML Metadata will be validated for accuracy when loading it. Obviously XML is defined by a DTD or XSD schema and so should follow that. You can turn off such validations by setting the persistence property **datanucleus.metadata.validate** to false when creating your PMF. Note that this only turns off the XML strictness validation, and *not* the checks on inconsistency of specification of relations etc.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.7.1 JPA XML MetaData

JPA XML Meta-Data Reference

JPA XML MetaData has the following format. Please refer to the JPA ORM XSD for precise details. What follows provides a reference guide to MetaData elements.

- entity-mappings
    - description
    - persistence-unit-metadata
        - xml-mapping-metadata-complete
    - package
    - schema
    - catalog
    - access
    - sequence-generator
    - table-generator
    - named-query
        - query
    - named-native-query
        - query
    - sql-result-set-mapping
        - entity-result
            - field-result
        - column-result
    - mapped-superclass
        - description
        - id-class
        - exclude-default-listeners
        - exclude-superclass-listeners
        - entity-listeners
            - entity-listener
                - pre-persist
                - post-persist

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

        - pre-remove
        - post-remove
        - pre-update
        - post-update
        - post-load
    - pre-persist
    - post-persist
    - pre-remove
    - post-remove
    - pre-update
    - post-update
    - post-load
    - attributes
        - Same elements as under <entity>-><attributes>
- entity
    - description
    - table
        - unique-constraint
            - column-name
    - secondary-table
        - primary-key-join-column
        - unique-constraint
            - column-name
    - primary-key-join-column
    - id-class
    - inheritance
    - discriminator-value
    - discriminator-column
    - sequence-generator
    - table-generator
    - named-query
        - query
    - named-native-query
        - query

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- sql-result-set-mapping
    - entity-result
        - field-result
    - column-result
- exclude-default-listeners
- exclude-superclass-listeners
- entity-listeners
    - entity-listener
        - pre-persist
        - post-persist
        - pre-remove
        - post-remove
        - pre-update
        - post-update
        - post-load
- pre-persist
- post-persist
- pre-remove
- post-remove
- pre-update
- post-update
- post-load
- attribute-override
    - column
- association-override
    - join-column
- attributes
    - id
        - column
        - generated-value
        - sequence-generator
        - table-generator
    - embedded-id
    - basic

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

        - column
        - lob
        - temporal
        - enumerated
    - version
        - column
    - many-to-one
        - join-column
        - join-table
            - join-column
            - inverse-join-column
            - unique-constraint
                - column-name
        - cascade
            - cascade-all
            - cascade-persist
            - cascade-merge
            - cascade-remove
            - cascade-refresh
    - one-to-many
        - order-by
        - order-column
        - map-key
        - join-table
            - join-column
            - inverse-join-column
            - unique-constraint
                - column-name
        - join-column
        - cascade
            - cascade-all
            - cascade-persist
            - cascade-merge
            - cascade-remove

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

            - cascade-refresh
        - one-to-one
            - join-column
            - join-table
                - join-column
                - inverse-join-column
                - unique-constraint
                    - column-name
            - cascade
                - cascade-all
                - cascade-persist
                - cascade-merge
                - cascade-remove
                - cascade-refresh
        - many-to-many
            - order-by
            - order-column
            - map-key
            - join-table
                - join-column
                - inverse-join-column
                - unique-constraint
                    - column-name
            - cascade
                - cascade-all
                - cascade-persist
                - cascade-merge
                - cascade-remove
                - cascade-refresh
        - transient
- embeddable
    - embeddable-attributes
        - basic

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- transient

Metadata for description tag

The <description> element (<entity-mappings>) contains the text describing all classes (and hence entities) defined in this file. It serves no useful purpose other than descriptive.

Metadata for xml-mapping-metadata-complete tag

The <xml-mapping-metadata-complete> element (under <persistence-unit-metadata>) when specified defines that the classes in this file are fully specified with just their metadata and that any annotations should be ignored.

Metadata for package tag

The <package> element (under <entity-mappings>) contains the text defining the package into which all classes in this file belong.

Metadata for schema tag

The <schema> element (under <entity-mappings>) contains the default schema for all classes in this file.

Metadata for catalog tag

The <catalog> element (under <entity-mappings>) contains the default catalog for all classes in this file.

Metadata for access tag

The <access> element (under <entity-mappings>) contains the setting for how to access the persistent fields/properties. This can be set to either "FIELD" or "PROPERTY".

Metadata for sequence-generator tag

The <sequence-generator> element (under <entity-mappings>, or <entity> or <id>) defines a generator of sequence values, for use elsewhere in this persistence-unit.

Attribute	Description	Values
name	Name of the generator (required)	
sequence-name	Name of the sequence	
initial-value	Initial value for the sequence	1
allocation-size	Number of values that the sequence allocates when needed	50

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for table-generator tag

The <table-generator> element (under <entity-mappings>, or <entity> or <id>) defines a generator of sequence values using a datastore table, for use elsewhere in this persistence-unit.

Attribute	Description	Values
name	Name of the generator (required)	
table	name of the table to use for sequences	SEQUENCE_TABLE
catalog	Catalog to store the sequence table	
schema	Schema to store the sequence table	
pk-column-name	Name of the primary-key column in the table	SEQUENCE_NAME
value-column-name	Name of the value column in the table	NEXT_VAL
pk-column-value	Name of the value to use in the primary key column (for this row)	{name of the class}
initial-value	Initial value to use in the table	0
allocation-size	Number of values to allocate when needed	50

Metadata for named-query tag

The <named-query> element (under <entity-mappings> or under <entity>) defines a JPQL query that will be accessible at runtime via the name. The element itself will contain the text of the query. It has the following attributes

Attribute	Description	Values
name	Name of the query	

Metadata for named-native-query tag

The <named-native-query> element (under <entity-mappings> or under <entity>) defines an SQL query that will be accessible at runtime via the name. The element itself will contain the text of the query. It has the following attributes

Attribute	Description	Values
name	Name of the query	

Metadata for sql-result-set-mapping tag

The <sql-result-set-mapping> element (under <entity-mappings> or under <entity>) defines how the results of the SQL query are output to the user per row of the result set. It will contain sub-elements. It

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

has the following attributes

Attribute	Description	Values
name	Name of the SQL result-set mapping (referenced by native queries)	

Metadata for entity-result tag

The <entity-result> element (under <sql-result-set-mapping>) defines an entity that is output from an SQL query per row of the result set. It can contain sub-elements of type <field-result>. It has the following attributes

Attribute	Description	Values
entity-class	Class of the entity	
discriminator-column	Column containing any discriminator (so subclasses of the entity type can be distinguished)	

Metadata for field-result tag

The <field-result> element (under <entity-result>) defines a field of an entity and the column representing it in an SQL query. It has the following attributes

Attribute	Description	Values
name	Name of the entity field	
column	Name of the SQL column	

Metadata for column-result tag

The <column-result> element (under <sql-result-set-mapping>) defines a column that is output directly from an SQL query per row of the result set. It has the following attributes

Attribute	Description	Values
name	Name of the SQL column	

Metadata for mapped-superclass tag

These are attributes within the <mapped-superclass> tag (under <entity-mappings>). This is used to define the persistence definition for a class that has no table but is mapped.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
class	Name of the class (required)	
metadata-complete	Whether the definition of persistence of this class is complete with this MetaData definition. That is, should any annotations be ignored.	true | **false**

Metadata for entity tag

These are attributes within the <entity> tag (under <entity-mappings>). This is used to define the persistence definition for this class.

Attribute	Description	Values
class	Name of the class (required)	
name	Name of the entity. Used by JPQL queries	
metadata-complete	Whether the definition of persistence of this class is complete with this MetaData definition. That is, should any annotations be ignored.	true | **false**

Metadata for description tag

The <description> element (under <entity>) contains the text describing the class being persisted. It serves no useful purpose other than descriptive.

Metadata for table tag

These are attributes within the <table> tag (under <entity>). This is used to define the table where this class will be persisted.

Attribute	Description	Values
name	Name of the table	
catalog	Catalog where the table is stored	
schema	Schema where the table is stored	

Metadata for secondary-table tag

These are attributes within the <secondary-table> tag (under <entity>). This is used to define the join of a secondary table back to the primary table where this class will be persisted.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
name	Name of the table	
catalog	Catalog where the table is stored	
schema	Schema where the table is stored	

Metadata for join-table tag

These are attributes within the <join-table> tag (under <one-to-one>, <one-to-many>, <many-to-many>). This is used to define the join table where a collection/maps relationship will be persisted.

Attribute	Description	Values
name	Name of the join table	
catalog	Catalog where the join table is stored	
schema	Schema where the join table is stored	

Metadata for unique-constraint tag

This element is specified under the <table>, <secondary-table> or <join-table> tags. This is used to define a unique constraint on the table. No attributes are provided, just sub-element(s) "column-name"

Metadata for column tag

These are attributes within the <column> tag (under <basic>). This is used to define the column where the data will be stored.

Attribute	Description	Values
name	Name of the column	
unique	Whether the column is unique	true | **false**
nullable	Whether the column is nullable	**true** | false
insertable	Whether the column is insertable	**true** | false
updatable	Whether the column is updatable	**true** | false
column-definition	Some vague JPA term that you put anything in and get any unexpected results from	
table	Table for the column ?	
length	Length for the column (when string type)	255
precision	Precision for the column (when numeric type)	0

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
scale	Scale for the column (when numeric type)	0

Metadata for primary-key-join-column tag

These are attributes within the <primary-join-key-column> tag (under <secondary-table> or <entity>). This is used to define the join of PK columns between secondary and primary tables, or between table of subclass and table of base class.

Attribute	Description	Values
name	Name of the column	
referenced-column-name	Name of column in primary table	

Metadata for join-column tag

These are attributes within the <join-column> tag (under <join-table>). This is used to define the join column.

Attribute	Description	Values
name	Name of the column	
referenced-column-name	Name of the column at the other side of the relation that this is a FK to	
unique	Whether the column is unique	true | **false**
nullable	Whether the column is nullable	**true** | false
insertable	Whether the column is insertable	**true** | false
updatable	Whether the column is updatable	**true** | false
column-definition	Some vague JPA term that you put anything in and get any unexpected results from. Not supported by DataNucleus.	
table	Table for the column ?	

Metadata for inverse-join-column tag

These are attributes within the <inverse-join-column> tag (under <join-table>). This is used to define the join column to the element.

Attribute	Description	Values
name	Name of the column	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
referenced-column-name	Name of the column at the other side of the relation that this is a FK to	
unique	Whether the column is unique	true | **false**
nullable	Whether the column is nullable	**true** | false
insertable	Whether the column is insertable	**true** | false
updatable	Whether the column is updatable	**true** | false
column-definition	Some vague JPA term that you put anything in and get any unexpected results from. Not supported by DataNucleus.	
table	Table for the column ?	

Metadata for id-class tag

These are attributes within the <id-class> tag (under <entity>). This defines a identity class to be used for this entity.

Attribute	Description	Values
class	Name of the identity class (required)	

Metadata for inheritance tag

These are attributes within the <inheritance> tag (under <entity>). This defines the inheritance of the class.

Attribute	Description	Values
strategy	Strategy for inheritance in terms of storing this class	SINGLE_TABLE | JOINED | TABLE_PER_CLASS

Metadata for discriminator-value tag

These are attributes within the <discriminator-value> tag (under <entity>). This defines the value used in a discriminator. The value is contained in the element. Specification of the value will result in a "value-map" discriminator strategy being adopted. If no discriminator-value is present, but discriminator-column is then "class-name" discriminator strategy is used.

Metadata for discriminator-column tag

These are attributes within the <discriminator-column> tag (under <entity>). This defines the column used for a discriminator.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
name	Name of the discriminator column	DTYPE
discriminator-type	Type of data stored in the discriminator column	STRING | CHAR | INTEGER
length	Length of the discriminator column	

Metadata for id tag

These are attributes within the <id> tag (under <attributes>). This is used to define the field used to be the identity of the class.

Attribute	Description	Values
name	Name of the field (required)	

Metadata for generated-value tag

These are attributes within the <generated-value> tag (under <id>). This is used to define how to generate the value for the identity field.

Attribute	Description	Values
strategy	Generation strategy. Please refer to the Identity Generation Guide	**auto** | identity | sequence | table
generator	Name of the generator to use if wanting to override the default DataNucleus generator for the specified strategy. Please refer to the <sequence-generator> and <table-generator>	

Metadata for embedded-id tag

These are attributes within the <embedded-id> tag (under <attributes>). This is used to define the field used to be the (embedded) identity of the class.

Attribute	Description	Values
name	Name of the field (required)	

Metadata for version tag

These are attributes within the <version> tag (under <attributes>). This is used to define the field used to be hold the version of the class.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Description	Values
name	Name of the field (required)	

Metadata for basic tag

These are attributes within the <basic> tag (under <attributes>). This is used to define the persistence information for the field.

Attribute	Description	Values
name	Name of the field (required)	
fetch	Fetch type for this field	LAZY | EAGER
optional	Whether this field may be null and may be used in schema generation	true | false

Metadata for temporal tag

These are attributes within the <temporal> tag (under <basic>). This is used to define the details of persistence as a temporal type. The contents of the element can be one of DATE, TIME, TIMESTAMP.

Metadata for enumerated tag

These are attributes within the <enumerated> tag (under <basic>). This is used to define the details of persistence as an enum type. The contents of the element can be one of **ORDINAL** or STRING to represent whether the enum is persisted as an integer-based or the actual string.

Metadata for one-to-one tag

These are attributes within the <one-to-one> tag (under <attributes>). This is used to define that the field is part of a 1-1 relation.

Attribute	Description	Values
name	Name of the field (required)	
target-entity	Class name of the related entity	
fetch	Whether the field should be fetched immediately	**EAGER** | LAZY
optional	Whether the field can store nulls.	**true** | false
mapped-by	Name of the field that owns the relation (specified on the inverse side)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for many-to-one tag

These are attributes within the <many-to-one> tag (under <attributes>). This is used to define that the field is part of a N-1 relation.

Attribute	Description	Values
name	Name of the field (required)	
target-entity	Class name of the related entity	
fetch	Whether the field should be fetched immediately	**EAGER** | LAZY
optional	Whether the field can store nulls.	**true** | false

Metadata for one-to-many tag

These are attributes within the <one-to-many> tag (under <attributes>). This is used to define that the field is part of a 1-N relation.

Attribute	Description	Values
name	Name of the field (required)	
target-entity	Class name of the related entity	
fetch	Whether the field should be fetched immediately	EAGER | **LAZY**
mapped-by	Name of the field that owns the relation (specified on the inverse side)	

Metadata for many-to-many tag

These are attributes within the <many-to-many> tag (under <attributes>). This is used to define that the field is part of a M-N relation.

Attribute	Description	Values
name	Name of the field (required)	
target-entity	Class name of the related entity	
fetch	Whether the field should be fetched immediately	EAGER | **LAZY**
mapped-by	Name of the field on the non-owning side that completes the relation. Specified on the owner side	

Metadata for order-by tag

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This element is specified under <one-to-many> or <many-to-many>. It is used to define the field(s) of the element class that is used for ordering the elements when they are retrieved from the datastore. It has no attributes and the ordering is specified within the element itself. It should be a comma-separated list of field names (of the element) with optional "ASC" or "DESC" to signify ascending or descending

Metadata for order-column tag

This element is specified under <one-to-many> or <many-to-many>. It is used to define that the List will be ordered with the ordering stored in a surrogate column in the other table.

Attribute	Description	Values
name	Name of the column	{fieldName}_ORDER
nullable	Whether the column is nullable	**true** | false
insertable	Whether the column is insertable	**true** | false
updatable	Whether the column is updatable	**true** | false
column-definition	Some vague JPA term that you put anything in and get any unexpected results from	
base	Origin of the ordering (value for the first element)	0

Metadata for map-key tag

These are attributes within the <map-key> tag (under <one-to-many> or <many-to-many>). This is used to define the field of the value class that is the key of a Map.

Attribute	Description	Values
name	Name of the field (required)	

Metadata for transient tag

These are attributes within the <transient> tag (under <attributes>). This is used to define that the field is not to be persisted.

Attribute	Description	Values
name	Name of the field (required)	

Metadata for exclude-default-listeners tag

This element is specified under <mapped-superclass> or <entity> and is used to denote that any default listeners defined in this file will be ignored.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for exclude-superclass-listeners tag

This element is specified under <mapped-superclass> or <entity> and is used to denote that any listeners of superclasses will be ignored.

Metadata for entity-listener tag

These are attributes within the <entity-listener> tag (under <entity-listeners>). This is used to an EntityListener class and the methods it uses

Attribute	Description	Values
class	Name of the EntityListener class that receives the callbacks for this Entity	

Metadata for pre-persist tag

These are attributes within the <pre-persist> tag (under <entity>). This is used to define any "PrePersist" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for post-persist tag

These are attributes within the <post-persist> tag (under <entity>). This is used to define any "PostPersist" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for pre-remove tag

These are attributes within the <pre-remove> tag (under <entity>). This is used to define any "PreRemove" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for post-remove tag

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

These are attributes within the <post-remove> tag (under <entity>). This is used to define any "PostRemove" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for pre-update tag

These are attributes within the <pre-remove> tag (under <entity>). This is used to define any "PreUpdate" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for post-update tag

These are attributes within the <post-update> tag (under <entity>). This is used to define any "PostUpdate" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for post-load tag

These are attributes within the <post-load> tag (under <entity>). This is used to define any "PostLoad" method callback.

Attribute	Description	Values
method-name	Name of the method (required)	

Metadata for attribute-override tag

These are attributes within the <attribute-override> tag (under <entity>). This is used to override the columns for any fields in superclasses

Attribute	Description	Values
name	Name of the field/property (required)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Metadata for association-override tag

These are attributes within the <association-override> tag (under <entity>). This is used to override the columns for any N-1/1-1 fields in superclasses

Attribute	Description	Values
name	Name of the field/property (required)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.7.2 JPA Annotations

JPA Annotations

One of the things that JDK 1.5 provides that can be of some use to DataNucleus is annotations. DataNucleus supports both JPA1 and JDO2 annotations. In this section we will document some of the more important JPA1 annotations. When selecting to use annotations please bear in mind the following :-

- You must be using **JDK 1.5 or above**. Annotations are not available in earlier JDKs
- You must have the **DataNucleus Java5 plugin** available in your CLASSPATH.
- Annotations should really only be used for attributes of persistence that you won't be changing at deployment. Things such as table and column names shouldn't really be specified using annotations although it is permitted. Instead it would be better to put such information in an ORM file.
- Annotations can be added in two places - for the class as a whole, or for a field in particular.
- You can annotate fields or getters with field-level information. It doesn't matter which.
- Annotations are prefixed by the @ symbol and can take properties (in brackets after the name, comma-separated)
- JPA doesn't provide for some key JDO concepts and DataNucleus provides its own annotations for these cases.
- You have to import "javax.persistence.XXX" where XXX is the annotation name of a JPA annotation
- You have to import "org.datanucleus.annotations.jpa.XXX" where XXX is the annotation name of a DataNucleus value-added annotation

Annotations supported by DataNucleus are shown below. Not all have their documentation written yet. The annotations/attributes coloured in pink are ORM and really should be placed in XML rather than directly in the class using annotations. The annotations coloured in blue are DataNucleus extensions and should be used only where you dont mind losing implementation-independence.

Annotation	Class/Field	Description
@Entity	Class	Specifies that the class is persistent
@MappedSuperclass	Class	Specifies that this class contains persistent information to be mapped
@PersistenceAware	Class	Specifies that the class is not persistent but needs to be able to access fields of persistent classes
@Embeddable	Class	Specifies that the class is persistent embedded in another persistent class
@IdClass	Class	Defines the primary key class for this class
@DatastoreIdentity	Class	Defines a class as using datastore-identity (DataNucleus extension).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Annotation	Class/Field	Description
@EntityListeners	Class	Specifies class(es) that are listeners for events from instances of this class
@NamedQueries	Class	Defines a series of named (JPQL) queries for use in the current persistence unit
@NamedQuery	Class	Defines a named (JPQL) query for use in the current persistence unit
@NamedNativeQuery	Class	Defines a named (SQL) query for use in the current persistence unit
@NamedNativeQueries	Class	Defines a series of named (SQL) queries for use in the current persistence unit
@SqlResultSetMapping	Class	Defines a result mapping for an SQL query for use in the current persistence unit
@SqlResultSetMappings	Class	Defines a series of mappings for SQL queries for use in the current persistence unit
@Inheritance	Class	Specifies the inheritance model for persisting this class
@Table	Class	Defines the table where this class will be stored
@SecondaryTable	Class	Defines a secondary table where some fields of this class will be stored
@DiscriminatorColumn	Class	Defines the column where any discriminator will be stored
@DiscriminatorValue	Class	Defines the value to be used in the discriminator for objects of this class
@PrimaryKeyJoinColumns	Class	Defines the names of the PK columns when this class has a superclass
@PrimaryKeyJoinColumn	Class	Defines the name of the PK column when this class has a superclass
@AttributeOverride	Class	Defines a field in a superclass that will have its column overridden
@AttributeOverrides	Class	Defines the field(s) of superclasses that will have their columns overridden
@AssociationOverride	Class	Defines a N-1/1-1 field in a superclass that will have its column overridden
@AssociationOverrides	Class	Defines the N-1/1-1 field(s) of superclasses that will have their columns overridden
@SequenceGenerator	Class/Field/Method	Defines a generator of values using sequences in the datastore for use with persistent entities
@TableGenerator	Class/Field/Method	Defines a generator of sequences using a table in the datastore for use with persistent entities
@Embedded	Field/Method	Defines this field as being embedded
@Id	Field/Method	Defines this field as being (part of) the identity for the class

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Annotation	Class/Field	Description
@EmbeddedId	Field/Method	Defines this field as being (part of) the identity for the class, and being embedded into this class
@Version	Field/Method	Defines this field as storing the version for the class
@Basic	Field/Method	Defines this field as being persistent
@Transient	Field/Method	Defines this field as being transient (not persisted)
@OneToOne	Field/Method	Defines this field as being a 1-1 relation with another persistent entity
@OneToMany	Field/Method	Defines this field as being a 1-N relation with other persistent entities
@ManyToMany	Field/Method	Defines this field as being a M-N relation with other persistent entities
@ManyToOne	Field/Method	Defines this field as being a N-1 relation with another persistent entity
@ElementCollection	Field/Method	Defines this field as being a 1-N relation of Objects that are not Entities. **This is new from JPA2**
@GeneratedValue	Field/Method	Defines that this field has its value generated using a generator
@MapKey	Field/Method	Defines that this field is the key to a map
@OrderBy	Field/Method	Defines the field(s) used for ordering the elements in this collection
@OrderColumn	Field/Method	Defines that ordering should be attributed by the implementation using a surrogate column.**This is new from JPA2**
@PrePersist	Field/Method	Defines this method as being a callback for pre-persist events
@PostPersist	Field/Method	Defines this method as being a callback for post-persist events
@PreRemove	Field/Method	Defines this method as being a callback for pre-remove events
@PostRemove	Field/Method	Defines this method as being a callback for post-remove events
@PreUpdate	Field/Method	Defines this method as being a callback for pre-update events
@PostUpdate	Field/Method	Defines this method as being a callback for post-update events
@PostLoad	Field/Method	Defines this method as being a callback for post-load events
@JoinTable	Field/Method	Defines this field as being stored using a join table
@CollectionTable	Field/Method	Defines this field as being stored using a join table when containing non-entity objects. **This is new from JPA2**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Annotation	Class/Field	Description
@Lob	Field/Method	Defines this field as being stored as a large object
@Temporal	Field/Method	Defines this field as storing temporal data
@Enumerated	Field/Method	Defines this field as storing enumerated data
@Column	Field/Method	Defines the column where this field is stored
@JoinColumn	Field/Method	Defines a column for joining to either a join table or foreign key relation
@JoinColumns	Field/Method	Defines the columns for joining to either a join table or foreign key relation (1-1, 1-N, N-1)
@Extensions	Class/Field/Method	Defines a series of DataNucleus extensions
@Extension	Class/Field/Method	Defines a DataNucleus extension

@Entity

This annotation is used when you want to mark a class as persistent. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the entity (used in JPQL to refer to the class)	

```
@Entity
public class MyClass
{
    ...
}
```

@MappedSuperclass

This annotation is used when you want to mark a class as persistent but without a table of its own and being the superclass of the class that has a table, meaning that all of its fields are persisted into the table of its subclass. Specified on the **class**.

```
@MappedSuperclass
public class MyClass
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@PersistenceAware

This annotation is used when you want to mark a class as knowing about persistence but not persistent itself. That is, it manipulates the fields of a persistent class directly rather than using accessors. **This is a DataNucleus extension**. Specified on the **class**.

```
@PersistenceAware
public class MyClass
{
    ...
}
```

@Embeddable

This annotation is used when you want to mark a class as persistent and only storable embedded in another object. Specified on the **class**.

```
@Embeddable
public class MyClass
{
    ...
}
```

@Inheritance

This annotation is used to define the inheritance persistence for this class. Specified on the **class**.

Attribute	Type	Description	Default
strategy	InheritanceType	Inheritance strategy	**SINGLE_TABLE** | JOINED | TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class MyClass
{
    ...
}
```

@Table

This annotation is used to define the table where objects of a class will be stored. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	Name of the table	
catalog	String	Name of the catalog	
schema	String	Name of the schema	
uniqueConstraints	UniqueConstraint[]	Any unique constraints to apply to the table	

```
@Entity
@Table(name="MYTABLE", schema="PUBLIC")
public class MyClass
{
    ...
}
```

@SecondaryTable

This annotation is used to define a secondary table where some fields of this class are stored in another table. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the table	
catalog	String	Name of the catalog	
schema	String	Name of the schema	
pkJoinColumns	PrimaryKeyJoinColumns[]	Join columns for the PK of the secondary table back to the primary table	
uniqueConstraints	UniqueConstraint[]	Any unique constraints to apply to the table	

```
@Entity
@Table(name="MYTABLE", schema="PUBLIC")
@SecondaryTable(name="MYOTHERTABLE", schema="PUBLIC",
columns={@PrimaryKeyJoinColumn(name="MYCLASS_ID")})
public class MyClass
{
    ...
}
```

@IdClass

This annotation is used to define a primary-key class for the identity of this class. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
value	Class	Identity class	

```
@Entity
@IdClass(org.datanucleus.samples.MyIdentity.class)
public class MyClass
{
    ...
}
```

@DatastoreIdentity

This DataNucleus-extension annotation is used to define that the class uses datastore-identity. Specified on the **class**.

Attribute	Type	Description	Default
generationType	GenerationType	Strategy to use when generating the values for this field. Has possible values of GenerationType TABLE, SEQUENCE, IDENTITY, AUTO.	**AUTO** | TABLE | SEQUENCE
generator	String	Name of the generator to use. See @TableGenerator and @SequenceGenerator	
column	String	Name of the column for persisting the datastore identity value	

```
@Entity
@DatastoreIdentity(column="MY_ID")
public class MyClass
{
    ...
}
```

@EntityListeners

This annotation is used to define a class or classes that are listeners for events from instances of this class. Specified on the **class**.

Attribute	Type	Description	Default
value	Class[]	Entity listener class(es)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Entity
@EntityListeners(org.datanucleus.MyListener.class)
public class MyClass
{
    ...
}
```

@NamedQueries

This annotation is used to define a series of named (JPQL) queries that can be used in this persistence unit. Specified on the **class**.

Attribute	Type	Description	Default
value	NamedQuery[]	The named queries	

```
@Entity
@NamedQueries({@NamedQuery(name="AllPeople", query="SELECT p FROM Person p"),
               @NamedQuery(name="PeopleCalledJones", query="SELECT p FROM Person p
WHERE p.surname = 'Jones'")})
public class Person
{
    ...
}
```

@NamedQuery

This annotation is used to define a named (JPQL) query that can be used in this persistence unit. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Symbolic name for the query. The query will be referred to under this name	
query	String	The JPQL query	

```
@Entity
@NamedQuery(name="AllPeople", query="SELECT p FROM Person p")
public class Person
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@NamedNativeQueries

This annotation is used to define a series of named native (SQL) queries that can be used in this persistence unit. Specified on the **class**.

Attribute	Type	Description	Default
value	NamedNativeQuery[]	The named native queries	

```
@Entity
@NamedNativeQueries({@NamedNativeQuery(name="AllPeople", query="SELECT * FROM PERSON
WHERE SURNAME = 'Smith'"),
     @NamedNativeQuery(name="PeopleCalledJones", query="SELECT * FROM PERSON WHERE
SURNAME = 'Jones')})
public class Person
{
    ...
}
```

@NamedNativeQuery

This annotation is used to define a named (SQL) query that can be used in this persistence unit. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Symbolic name for the query. The query will be referred to under this name	
query	String	The SQL query	
resultClass	Class	Class into which the result rows will be placed	void.class

```
@Entity
@NamedNativeQuery(name="PeopleCalledSmith", query="SELECT * FROM PERSON WHERE
SURNAME = 'Smith'")
public class Person
{
    ...
}
```

@SqlResultSetMappings

This annotation is used to define a series of result mappings for SQL queries that can be used in this persistence unit. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
value	SqlResultSetMapping[]	The SQL result mappings	

```
@Entity
@SqlResultSetMappings({
    @SqlResultSetMapping(name="PEOPLE_PLUS_AGE",
        entities={@EntityResult(entityClass=Person.class)},
columns={@ColumnResult(name="AGE")}),
    @SqlResultSetMapping(name="FIRST_LAST_NAMES",
        columns={@ColumnResult(name="FIRSTNAME"), @ColumnResult(name="LASTNAME")})
    })
public class Person
{
    ...
}
```

@SqlResultSetMapping

This annotation is used to define a mapping for the results of an SQL query and can be used in this persistence unit. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Symbolic name for the mapping. The mapping will be referenced under this name	
entities	EntityResult[]	Set of entities extracted from the SQL query	
columns	ColumnResult[]	Set of columns extracted directly from the SQL query	

```
@Entity
@SqlResultSetMapping(name="PEOPLE_PLUS_AGE",
    entities={@EntityResult(entityClass=Person.class)},
columns={@ColumnResult(name="AGE")})
public class Person
{
    ...
}
```

@PrePersist

This annotation is used to define a method that is a callback for pre-persist events. Specified on the **method**. It has no attributes.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Entity
public class MyClass
{
    ...

    @PrePersist
    void registerObject()
    {
        ...
    }
}
```

@PostPersist

This annotation is used to define a method that is a callback for post-persist events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PostPersist
    void doSomething()
    {
        ...
    }
}
```

@PreRemove

This annotation is used to define a method that is a callback for pre-remove events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PreRemove
    void registerObject()
    {
        ...
    }
}
```

@PostRemove

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This annotation is used to define a method that is a callback for post-remove events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PostRemove
    void doSomething()
    {
        ...
    }
}
```

@PreUpdate

This annotation is used to define a method that is a callback for pre-update events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PreUpdate
    void registerObject()
    {
        ...
    }
}
```

@PostUpdate

This annotation is used to define a method that is a callback for post-update events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PostUpdate
    void doSomething()
    {
        ...
    }
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@PostLoad

This annotation is used to define a method that is a callback for post-load events. Specified on the **method**. It has no attributes.

```
@Entity
public class MyClass
{
    ...

    @PostLoad
    void registerObject()
    {
        ...
    }
}
```

@SequenceGenerator

This annotation is used to define a generator using sequences in the datastore. It is scoped to the persistence unit. Specified on the **class/field/method**.

Attribute	Type	Description	Default
name	String	Name for the generator (required)	
sequenceName	String	Name of the underlying sequence that will be used	
initialValue	int	Initial value for the sequence (optional)	1
allocationSize	int	Number of values to be allocated each time (optional)	50

```
@Entity
@SequenceGenerator(name="MySeq", sequenceName="SEQ_2")
public class MyClass
{
    ...
}
```

@TableGenerator

This annotation is used to define a generator using a table in the datastore for storing the values. It is scoped to the persistence unit. Specified on the **class/field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	Name for the generator (required)	
table	String	Name of the table to use	SEQUENCE_TABLE
catalog	String	Catalog of the table to use	
schema	String	Schema of the table to use	
pkColumnName	String	Name of the primary key column for the table	SEQUENCE_NAME
valueColumnName	String	Name of the value column for the table	NEXT_VAL
pkColumnValue	String	Value to store in the PK column for the row used by this generator	{name of the class}
initialValue	int	Initial value for the table row (optional)	0
allocationSize	int	Number of values to be allocated each time (optional)	50

```
@Entity
@TableGenerator(name="MySeq", table="MYAPP_IDENTITIES", pkColumnValue="MyClass")
public class MyClass
{
    ...
}
```

@DiscriminatorColumn

This annotation is used to define the discriminator column for a class. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the discriminator column	DTYPE
discriminatorType	DiscriminatorType	Type of the discriminator column	STRING | CHAR | INTEGER
length	String	Length of the discriminator column	31

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
public class MyClass
{
    ...
}
```

@DiscriminatorValue

This annotation is used to define the value to be stored in the discriminator column for a class (when used). Specified on the **class**.

Attribute	Type	Description	Default
value	String	Value for the discriminator column	

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("MyClass")
public class MyClass
{
    ...
}
```

@PrimaryKeyJoinColumns

This annotation is used to define the names of the primary key columns when this class has a superclass. Specified on the **class**.

Attribute	Type	Description	Default
value	PrimaryKeyJoinColumn[]	Array of column definitions for the primary key	

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@PrimaryKeyJoinColumns({@PrimaryKeyJoinColumn(name="PK_FIELD_1",
referredColumnName="BASE_1_ID"),
                        @PrimaryKeyJoinColumn(name="PK_FIELD_2",
referredColumnName="BASE_2_ID")})
public class MyClass
{
    ...
}
```

@PrimaryKeyJoinColumn

This annotation is used to define the name of the primary key column when this class has a superclass. Specified on the **class**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
name	String	Name of the column	
referencedColumnName	String	Name of the associated PK column in the superclass	
columnDefinition	String	Some vague JPA attribute that might mean something to each JPA implementation, and its basically SQL. Ignored by DataNucleus	

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@PrimaryKeyJoinColumn(name="PK_FIELD_1")
public class MyClass
{
    ...
}
```

@AttributeOverride

This annotation is used to define a field of a superclass that has its column overridden. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the field	
column	Column	Column information	

```
@Entity
@AttributeOverride(name="attr", column=@Column(name="NEW_NAME"))
public class MyClass extends MySuperClass
{
    ...
}
```

@AttributeOverrides

This annotation is used to define fields of a superclass that have their columns overridden. Specified on the **class**.

Attribute	Type	Description	Default
value	AttributeOverride[]	The overrides	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Entity
@AttributeOverrides({@AttributeOverride(name="attr1",
column=@Column(name="NEW_NAME_1")),
                     @AttributeOverride(name="attr2",
column=@Column(name="NEW_NAME_2"))})
public class MyClass extends MySuperClass
{
    ...
}
```

@AssociationOverride

This annotation is used to define a 1-1/N-1 field of a superclass that has its column overridden. Specified on the **class**.

Attribute	Type	Description	Default
name	String	Name of the field	
joinColumn	JoinColumn	Column information for the FK column	

```
@Entity
@AssociationOverride(name="friend", joinColumn=@JoinColumn(name="FRIEND_ID"))
public class Employee extends Person
{
    ...
}
```

@AssociationOverrides

This annotation is used to define 1-1/N-1 fields of a superclass that have their columns overridden. Specified on the **class**.

Attribute	Type	Description	Default
value	AssociationOverride[]	The overrides	

```
@Entity
@AssociationOverrides({@AssociationOverride(name="friend",
joinColumn=@JoinColumn(name="FRIEND_ID")),
                       @AssociationOverride(name="teacher",
joinColumn=@JoinColumn(name="TEACHER_ID"))})
public class Employee extends Person
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@Id

This annotation is used to define a field to use for the identity of the class. Specified on the **field/method**.

```
@Entity
public class MyClass
{
    @Id
    long id;
    ...
}
```

@EmbeddedId

This annotation is used to define a field to use for the identity of the class when embedded. Specified on the **field/method**.

```
@Entity
public class MyClass
{
    @EmbeddedId
    MyPrimaryKey pk;
    ...
}
```

@Version

This annotation is used to define a field as holding the version for the class. Specified on the **field/method**.

```
@Entity
public class MyClass
{
    @Id
    long id;

    @Version
    int ver;
    ...
}
```

@Basic

This annotation is used to define a field of the class as persistent. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
fetch	FetchType	Type of fetching for this field	LAZY | **EAGER**
optional	boolean	Whether this field having a value is optional (can it have nulls)	**true** | false

```
@Entity
public class Person
{
    @Id
    long id;

    @Basic(optional=false)
    String forename;
    ...
}
```

@Transient

This annotation is used to define a field of the class as not persistent. Specified on the **field/method**.

```
@Entity
public class Person
{
    @Id
    long id;

    @Transient
    String personalInformation;
    ...
}
```

@JoinTable

This annotation is used to define that a collection/map is stored using a join table. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name of the table	
catalog	String	Name of the catalog	
schema	String	Name of the schema	
joinColumns	JoinColumn[]	Columns back to the owning object (with the collection/map)	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
inverseJoinColumns	JoinColumn[]	Columns to the element object (stored in the collection/map)	
uniqueConstraints	UniqueConstraint[]	Any unique constraints to apply to the table	

```
@Entity
public class Person
{
    @OneToMany
    @JoinTable(name="PEOPLES_FRIENDS")
    Collection friends;
    ...
}
```

@CollectionTable

This annotation is used to define that a collection/map of non-entities is stored using a join table. **This is new from JPA2** Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name of the table	
catalog	String	Name of the catalog	
schema	String	Name of the schema	
joinColumns	JoinColumn[]	Columns back to the owning object (with the collection/map)	
uniqueConstraints	UniqueConstraint[]	Any unique constraints to apply to the table	

```
@Entity
public class Person
{
    @ElementCollection
    @CollectionTable(name="PEOPLES_FRIENDS")
    Collection<String> values;
    ...
}
```

@Lob

This annotation is used to define that a field will be stored using a large object in the datastore. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Entity
public class Person
{
    @Lob
    byte[] photo;
    ...
}
```

@Temporal

This annotation is used to define that a field is stored as a temporal type. Specified on the **field/method**.

Attribute	Type	Description	Default
value	TemporalType	Type for storage	DATE | TIME | TIMESTAMP

```
@Entity
public class Person
{
    @Temporal(TemporalType.TIMESTAMP)
    java.util.Date dateOfBirth;
    ...
}
```

@Enumerated

This annotation is used to define that a field is stored enumerated (not that it wasnt obvious from the type!). Specified on the **field/method**.

Attribute	Type	Description	Default
value	EnumType	Type for storage	**ORDINAL** | STRING

```
enum Gender {MALE, FEMALE};

@Entity
public class Person
{
    @Enumerated
    Gender gender;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@OneToOne

This annotation is used to define that a field represents a 1-1 relation. Specified on the **field/method**.

Attribute	Type	Description	Default
targetEntity	Class	Class at the other side of the relation	
fetch	FetchType	Whether the field should be fetched immediately	**EAGER** | LAZY
optional	boolean	Whether the field can store nulls.	**true** | false
mappedBy	String	Name of the field that owns the relation (specified on the inverse side)	
cascade	CascadeType[]	Whether persist, update, delete, refresh operations are cascaded	

```
@Entity
public class Person
{
    @OneToOne
    Person bestFriend;
    ...
}
```

@OneToMany

This annotation is used to define that a field represents a 1-N relation. Specified on the **field/method**.

Attribute	Type	Description	Default
targetEntity	Class	Class at the other side of the relation	
fetch	FetchType	Whether the field should be fetched immediately	EAGER | **LAZY**
mappedBy	String	Name of the field that owns the relation (specified on the inverse side)	
cascade	CascadeType[]	Whether persist, update, delete, refresh operations are cascaded	

```
@Entity
public class Person
{
    @OneToMany
    Collection<Person> friends;
    ...
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
}
```

@ManyToMany

This annotation is used to define that a field represents a M-N relation. Specified on the **field/method**.

Attribute	Type	Description	Default
targetEntity	Class	Class at the other side of the relation	
fetch	FetchType	Whether the field should be fetched immediately	EAGER | **LAZY**
mappedBy	String	Name of the field on the non-owning side that completes the relation. Specified on the owner side.	
cascade	CascadeType[]	Whether persist, update, delete, refresh operations are cascaded	

```
@Entity
public class Customer
{
    @ManyToMany(mappedBy="customers")
    Collection<Supplier> suppliers;
    ...
}

@Entity
public class Supplier
{
    @ManyToMany
    Collection<Customer> customers;
    ...
}
```

@ManyToOne

This annotation is used to define that a field represents a N-1 relation. Specified on the **field/method**.

Attribute	Type	Description	Default
targetEntity	Class	Class at the other side of the relation	
fetch	FetchType	Whether the field should be fetched immediately	**EAGER** | LAZY
optional	boolean	Whether the field can store nulls.	**true** | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
cascade	CascadeType[]	Whether persist, update, delete, refresh operations are cascaded	

```
@Entity
public class House
{
    @OneToMany(mappedBy="house")
    Collection<Window> windows;
    ...
}

@Entity
public class Window
{
    @ManyToOne
    House house;
    ...
}
```

@ElementCollection

This annotation is used to define that a field represents a 1-N relation to non-entity objects. **This is new from JPA2** Specified on the **field/method**.

Attribute	Type	Description	Default
targetClass	Class	Class at the other side of the relation	
fetch	FetchType	Whether the field should be fetched immediately	EAGER | **LAZY**

```
@Entity
public class Person
{
    @ElementCollection
    Collection<String> values;
    ...
}
```

@GeneratedValue

This annotation is used to define the generation of a value for a (PK) field. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
strategy	GenerationType	Strategy to use when generating the values for this field. Has possible values of GenerationType TABLE, SEQUENCE, IDENTITY, AUTO.	GenerationType.AUTO
generator	String	Name of the generator to use. See @TableGenerator and @SequenceGenerator	

```
@Entity
public class Person
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}
```

@MapKey

This annotation is used to define the field in the value class that represents the key in a Map. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name of the field in the value class to use for the key. If no value is supplied and the field is a Map then it is assumed that the key will be the primary key of the value class. DataNucleus only supports this null value treatment if the primary key of the value has a single field.	

```
@Entity
public class Person
{
    @OneToMany
    @MapKey(name="nickname")
    Map<String, Person> friends;
    ...
}
```

@OrderBy

This annotation is used to define a field in the element class that is used for ordering the elements of the List when it is retrieved. Specified on the **field/method**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
value	String	Name of the field(s) in the element class to use for ordering the elements of the List when retrieving them from the datastore. This is used by JPA "ordered lists" as opposed to JDO "indexed lists" (which always return the elements in the same order as they were persisted. The value will be a comma separated list of fields and optionally have ASC/DESC to signify ascending or descending	

```
@Entity
public class Person
{
    @OneToMany
    @OrderBy(value="nickname")
    List<Person> friends;
    ...
}
```

@OrderColumn

This annotation is used to definethat the JPA implementation will handle the ordering of the List elements using a surrogate column. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name of the column to use.	*{fieldName}_ORDER*
nullable	boolean	Whether the column is nullable	**true** | false
insertable	boolean	Whether the column is insertable	**true** | false
updatable	boolean	Whether the column is updatable	**true** | false
base	int	Base for ordering (not currently supported)	0

```
@Entity
public class Person
{
    @OneToMany
    @OrderColumn
    List<Person> friends;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

@Column

This annotation is used to define the column where a field is stored. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name for the column	
unique	boolean	Whether the field is unique	true | **false**
nullable	boolean	Whether the field is nullable	**true** | false
insertable	boolean	Whether the field is insertable	**true** | false
updatable	boolean	Whether the field is updatable	**true** | false
table	String	Name of the table	
length	int	Length for the column	255
precision	int	Decimal precision for the column	0
scale	int	Decimal scale for the column	0

```
@Entity
public class Person
{
    @Basic
    @Column(name="SURNAME", length=100, nullable=false)
    String surname;
    ...
}
```

@JoinColumn

This annotation is used to define the FK column for joining to another table. This is part of a 1-1, 1-N, or N-1 relation. Specified on the **field/method**.

Attribute	Type	Description	Default
name	String	Name for the column	
referencedColumnName	String	Name of the column in the other table that this is the FK for	
unique	boolean	Whether the field is unique	true | **false**
nullable	boolean	Whether the field is nullable	**true** | false
insertable	boolean	Whether the field is insertable	**true** | false
updatable	boolean	Whether the field is updatable	**true** | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Attribute	Type	Description	Default
columnDefinition	String	Some vague JPA term that is some SQL to generate the column, but that will likely be different on all implementations. Not supported by DataNucleus.	

```
@Entity
public class Person
{
    @OneToOne
    @JoinColumn(name="PET_ID", nullable=true)
    Animal pet;
    ...
}
```

@JoinColumns

This annotation is used to define the FK columns for joining to another table. This is part of a 1-1, 1-N, or N-1 relation. Specified on the **field/method**.

Attribute	Type	Description	Default
value	JoinColumn[]	Details of the columns	

```
@Entity
public class Person
{
    @OneToOne
    @JoinColumns({@JoinColumn(name="PET1_ID"), @JoinColumn(name="PET2_ID")})
    Animal pet; // composite PK
    ...
}
```

@UniqueConstraint

This annotation is used to define a unique constraint to apply to a table. It is specified as part of @Table, @JoinTable or @SecondaryTable.

Attribute	Type	Description	Default
columnNames	String[]	Names of the column(s)	

```
@Entity
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Table(name="PERSON",
uniqueConstraints={@UniqueConstraint(columnNames={"firstName","lastName"})})
public class Person
{
    @Basic
    String firstName;

    @Basic
    String lastName;
    ...
}
```

@Extensions

DataNucleus Extension Annotation used to define a set of extensions specific to DataNucleus. Specified on the **class** or **field**.

Attribute	Type	Description	Default
value	Extension[]	Array of extensions - see @Extension annotation	

```
@Entity
@Extensions({@Extension(key="firstExtension", value="myValue"),
             @Extension(key="secondExtension", value="myValue")})
public class Person
{
    ...
}
```

@Extension

DataNucleus Extension Annotation used to define an extension specific to DataNucleus. Specified on the **class** or **field**.

Attribute	Type	Description	Default
vendorName	String	Name of the vendor	datanucleus
key	String	Key for the extension	
value	String	Value of the extension	

```
@Entity
@Extension(key="RunFast", value="true")
public class Person
{
    ...
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

6.8 Persistence Unit

Persistence Unit

When designing an application you can usually nicely separate your persistable objects into independent groupings that can be treated separately, perhaps within a different DAO object, if using DAOs. JPA1 introduces the idea of a *persistence-unit*. A *persistence-unit* provides a convenient way of specifying a set of metadata files, and classes, and jars that contain all classes to be persisted in a grouping. The persistence-unit is named, and the name is used for identifying it. Consequently this name can then be used when defining what classes are to be enhanced, for example.

To define a *persistence-unit* you first need to add a file **persistence.xml** to the *META-INF/* directory of your application jar. This file will be used to define your *persistence-unit*s. Let's show an example

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <!-- Online Store -->
    <persistence-unit name="OnlineStore">
        <provider>org.datanucleus.jpa.PersistenceProviderImpl</provider>
        <class>org.datanucleus.samples.metadata.store.Product</class>
        <class>org.datanucleus.samples.metadata.store.Book</class>
        <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
        <class>org.datanucleus.samples.metadata.store.Customer</class>
        <class>org.datanucleus.samples.metadata.store.Supplier</class>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.h2.Driver"/>
            <property name="datanucleus.ConnectionURL" value="jdbc:h2:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>

    <!-- Accounting -->
    <persistence-unit name="Accounting">
        <provider>org.datanucleus.jpa.PersistenceProviderImpl</provider>
<mapping-file>com/datanucleus/samples/metadata/accounts/orm.xml</mapping-file>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.h2.Driver"/>
            <property name="datanucleus.ConnectionURL" value="jdbc:h2:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>

</persistence>
```

In this example we have defined 2 *persistence-unit*s. The first has the name "OnlineStore" and contains 5 classes (annotated). The second has the name "Accounting" and contains a metadata file called "orm.xml"

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

in a particular package (which will define the classes being part of that unit). This means that once we have defined this we can reference these *persistence-unit*s in our persistence operations.

There are several sub-elements of this *persistence.xml* file

- **provider** - the JPA persistence provider to be used. **The JPA persistence "provider" for DataNucleus is *org.datanucleus.jpa.PersistenceProviderImpl***
- **jar-file** - name of a JAR file to scan for annotated classes to include in this persistence-unit.
- **mapping-file** - name of an XML "mapping" file containing persistence information to be included in this persistence-unit.
- **class** - name of an annotated class to include in this persistence-unit
- **properties** - properties defining the persistence factory to be used. Please refer to Persistence Properties Guide for details

Use with JPA1

JPA1 requires the "persistence-unit" name to be specified at runtime when creating the *EntityManagerFactory*, like this

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MyPersistenceUnit");
```

Restricting to specific classes

If you want to just have specific classes in the *persistence-unit* you can specify them using the **class** element, and then add **exclude-unlisted-classes**, like this

```
    <persistence-unit name="Store">
        <provider>org.datanucleus.jpa.PersistenceProviderImpl</provider>
        <class>org.datanucleus.samples.metadata.store.Product</class>
        <class>org.datanucleus.samples.metadata.store.Book</class>
        <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.h2.Driver"/>
            <property name="datanucleus.ConnectionURL" value="jdbc:h2:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>
```

If you don't include the **exclude-unlisted-classes** then DataNucleus will search for annotated classes starting at the *root* of the *persistence-unit* (the root directory in the CLASSPATH that contains the "META-INF/persistence.xml" file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.1 ORM with JPA

JPA Object/Relational Mapping

When you are using an RDBMS datastore you need to specify how your class will map on to the relational datastore. This part is termed **Object-Relational Mapping**. This is not required for other types of datastore. When you are persisting to RDBMS datastores you are mapping a series of objects into a series of datastore *tables* in a *schema*. These *tables* are interrelated using *foreign-keys*. With JPA1 you can define almost fully this object-relational mapping in the MetaData (or in annotations if you so wish).

The design of the persistence layer of an application requiring object-relational mapping can be approached in 3 ways.

- Forward Mapping - Here you have a set of model classes, and want to design the datastore schema that will store represent these classes.
- Reverse Mapping - Here you have an existing datastore schema, and want to design your model classes to represent this schema.
- Meet in the Middle Mapping - Here you have a set of model classes and you have an existing datastore schema, and you want to match them up.

DataNucleus can be used in all of these modes, though provides significant assistance for Forward Mapping cases. In particular, when using this mode you can use the DataNucleus SchemaTool to generate the datastore schema, based on a set of input classes and MetaData files. It should be noted though that DataNucleus SchemaTool also provides modes of operation for updating an existing schema, and hence can also be used for Meet in the Middle Mapping. Additionally, it can be used as a validation mechanism when designing your system in Reverse Mapping mode, where it will inform you of inconsistencies between your classes and your datastore schema.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.2 Schema Mapping

Schema Mapping

You saw in our basic class mapping guide how you define a classes basic persistence, notating which fields are persisted. The next step is to define how it maps to the schema of the datastore (in this case RDBMS). The simplest way of mapping is to map each class to its own table. This is the default model in JDO persistence (with the exception of inheritance). If you don't specify the table and column names, then DataNucleus will generate table and column names for you. **You should specify your table and column names if you have an existing schema.** Failure to do so will mean that DataNucleus uses its own names and these will almost certainly not match what you have in the datastore.

Tables and Column names

The main thing that developers want to do when they set up the persistence of their data is to control the names of the tables and columns used for storing the classes and fields. This is an essential step when mapping to an existing schema, because it is necessary to map the classes onto the existing database entities. Let's take an example

```
public class Hotel
{
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    ...
}
```

In our case we want to map this class to a table called ESTABLISHMENT, and has columns NAME, DIRECTION, PHONE and NUMBER_OF_ROOMS (amongst other things). So we define our Meta-Data like this

```
<entity class="Hotel">
    <table name="ESTABLISHMENT"/>
    <attributes>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="address">
            <column name="DIRECTION"/>
        </basic>
        <basic name="telephoneNumber">
            <column name="PHONE"/>
        </basic>
        <basic name="numberOfRooms">
            <column name="NUMBER_OF_ROOMS"/>
        </basic>
    </attributes>
</entity>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So we have defined the table and the column names. It should be mentioned that if you don't specify the table and column names then DataNucleus will generate names for the datastore identifiers consistent with the JPA specification. The table name will be based on the class name, and the column names will be based on the field names and the role of the field (if part of a relationship).

See also :-

- Identifier Guide - defining the identifiers to use for table/column names
- MetaData reference for <column> element

Column nullability and default values

So we've seen how to specify the basic structure of a table, naming the table and its columns, and how to control the types of the columns. We can extend this further to control whether the columns are allowed to contain nulls. Let's take a related class for our hotel. Here we have a class to model the payments made to the hotel.

```
public class Payment
{
    Customer customer;
    String bankTransferReference;
    String currency;
    double amount;
}
```

In this class we can model payments from a customer of an amount. Where the customer pays by bank transfer we can save the reference number. Since the bank transfer reference is optional we want that column to be nullable. So let's specify the MetaData for the class.

```
<entity class="Payment">
    <attributes>
        <one-to-one name="customer">
            <primary-key-join-column name="CUSTOMER_ID"/>
        </one-to-one>
        <basic name="bankTransferReference">
            <column name="TRANSFER_REF" nullable="true"/>
        </basic>
        <basic name="currency">
            <column name="CURRENCY" default-value="GBP"/>
        </basic>
        <basic name="amount">
            <column name="AMOUNT"/>
        </basic>
    </attributes>
</entity>
```

So we make use of the nullable attribute. The table, when created by DataNucleus, will then provide the nullability that we require. Unfortunately with JPA there is no way to specify a default value for a field when it hasnt been set (unlike JDO where you can do that).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

See also :-

- MetaData reference for <column> element

Column types

DataNucleus will provide a default type for any columns that it creates, but it will allow users to override this default. The default that DataNucleus chooses is always based on the Java type for the field being mapped. For example a Java field of type "int" will be mapped to a column type of INTEGER in RDBMS datastores. Similarly String will be mapped to VARCHAR. JPA does NOT allow detailed control over the JDBC type as such, with the exception of distinguishing BLOB/CLOB/TIME/TIMESTAMP types. The thing it does allow is permit control over the length/precision/scale of columns to we define this as follows

```
<entity name="Payment">
    <attributes>
        <one-to-one name="customer">
            <primary-key-join-column name="CUSTOMER_ID"/>
        </one-to-one>
        <basic name="bankTransferReference">
            <column name="TRANSFER_REF" nullable="true" length="255"/>
        </basic>
        <basic name="currency">
            <column name="CURRENCY" default-value="GBP" length="3"/>
        </basic>
        <basic name="amount">
            <column name="AMOUNT" precision="10" scale="2"/>
        </basic>
    </attributes>
</entity>
```

So we have defined TRANSFER_REF to use VARCHAR(255) column type, CURRENCY to use (VAR)CHAR(3) column type, and AMOUNT to use DECIMAL(10,2) column type.

See also :-

- Types Guide - defining mapping of Java types
- RDBMS Types Guide - defining mapping of Java types to JDBC/SQL types
- MetaData reference for <column> element

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.3 Datastore Identifiers

JPA Datastore Identifiers

A datastore identifier is a simple name of a database object, such as a column, table, index, or view, and is composed of a sequence of letters, digits, and underscores (_) that represents it's name. DataNucleus allows users to specify the names of tables, columns, indexes etc but if the user doesn't specify these DataNucleus will generate names. Generation of identifier names is controlled by an IdentifierFactory, and DataNucleus provides a default implementation. You can provide your own IdentifierFactory plugin to give your own preferred naming if so desired. You set the *IdentifierFactory* by setting the PMF property *datanucleus.identifierFactory*. Set it to the symbolic name of the factory you want to use. JPA defines what datastore identifiers should default to when not specified. DataNucleus provides a factory that meets this requirement.

- jpa IdentifierFactory (default for JPA persistence)

In describing the different possible naming conventions available out of the box with DataNucleus we'll use the following example

```
class MyClass
{
    String myField1;
    Collection<MyElement> elements1; // Using join table
    Collection<MyElement> elements2; // Using foreign-key
}

class MyElement
{
    String myElementField;
    MyClass myClass2;
}
```

IdentifierFactory 'jpa'

The *IdentifierFactory* "jpa" aims at providing a naming policy consistent with the "JPA1" specification.

Using the same example above, the rules in this *IdentifierFactory* mean that, assuming that the user doesnt specify any <column> elements :-

- *MyClass* will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MYFIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**
- *MyClass.elements1* will be persisted into a join table called **MYCLASS_MYELEMENT**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_MYCLASS_ID** (FK to owner table) and **ELEMENTS1_ELEMENT_ID** (FK to element table)

- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID** (FK to owner) table
- Any discriminator column will be called **DTYPE**
- Any index column in a List for field *MyClass.myField1* will be called **MYFIELD1_ORDER**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

IdentifierFactory - Controlling the Case

The underlying datastore will define what case of identifiers are accepted. By default, DataNucleus will capitalise names (assuming that the datastore supports it). You can however influence the case used for identifiers. This is specifiable with the PMF property *datanucleus.identifier.case*, having the following values

- UpperCase: identifiers are in upper case
- LowerCase: identifiers are in lower case
- PreserveCase: No case changes are made to the name of the identifier provided by the user (class name or jdo metadata).

Please be aware that some datastores only support UPPERCASE or lowercase identifiers and so setting this parameter may have no effect if your database doesn't support that option.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.4 Secondary Tables

JPA Secondary Tables

The standard JPA persistence strategy is to persist an object of a class into its own table. In some situations you may wish to map the class to a primary table as well as one or more secondary tables. For example when you have a Java class that could have been split up into 2 separate classes yet, for whatever reason, has been written as a single class, however you have a legacy datastore and you need to map objects of this class into 2 tables. JPA allows persistence of fields of a class into secondary tables.

The process for managing this situation is best demonstrated with an example. Let's suppose we have a class that represents a Printer. The Printer class contains within it various attributes of the toner cartridge. So we have

```
package com.mydomain.samples.secondarytable;

public class Printer
{
    long id;
    String make;
    String model;

    String tonerModel;
    int tonerLifetime;

    /**
     * Constructor.
     * @param make Make of printer (e.g Hewlett-Packard)
     * @param model Model of Printer (e.g LaserJet 1200L)
     * @param tonerModel Model of toner cartridge
     * @param tonerLifetime lifetime of toner (number of prints)
     */
    public Printer(String make, String model, String tonerModel, int tonerLifetime)
    {
        this.make = make;
        this.model = model;
        this.tonerModel = tonerModel;
        this.tonerLifetime = tonerLifetime;
    }

}
```

Now we have a database schema that has 2 tables (PRINTER and PRINTER_TONER) in which to store objects of this class. So we need to tell DataNucleus to perform this mapping. So we define the MetaData for the Printer class like this

```
<entity class="Printer">
    <table name="PRINTER"/>
    <secondary-table name="PRINTER_TONER">
        <primary-key-join-column name="PRINTER_REFID"/>
```

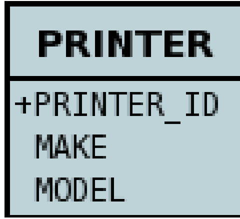
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

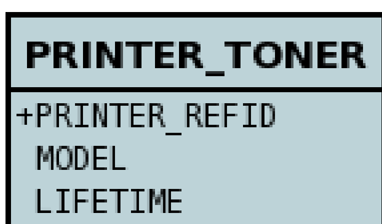
```
    </secondary-table>

    <attributes>
        <id name="id">
            <column name="PRINTER_ID"/>
        </id>
        <basic name="make">
            <column name="MAKE" length="40"/>
        </basic>
        <basic name="model">
            <column name="MODEL" length="100"/>
        </basic>
        <basic name="tonerModel">
            <column name="MODEL" table="PRINTER_TONER"/>
        </basic>
        <basic name="tonerLifetime">
            <column name="LIFETIME" table="PRINTER_TONER"/>
        </basic>
    </attributes>
</entity>
```

So here we have defined that objects of the Printer class will be stored in the primary table PRINTER. In addition we have defined that some fields are stored in the table PRINTER_TONER.

- We declare the "secondary-table"(s) that we will be using at the start of the definition.
- We define tonerModel and tonerLifetime to use columns in the table PRINTER_TONER. This uses the "table" attribute of <column>
- Whilst defining the secondary table(s) we will be using, we also define the join column to be called "PRINTER_REFID".

This results in the following database tables :-





So we now have our primary and secondary database tables. The primary key of the PRINTER_TONER table serves as a foreign key to the primary class. Whenever we persist a Printer object a row will be inserted into both of these tables.

See also :-

- MetaData reference for <secondary-table> element
- MetaData reference for <column> element
- Annotations reference for @SecondaryTable
- Annotations reference for @Column

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.5 Serialised Objects

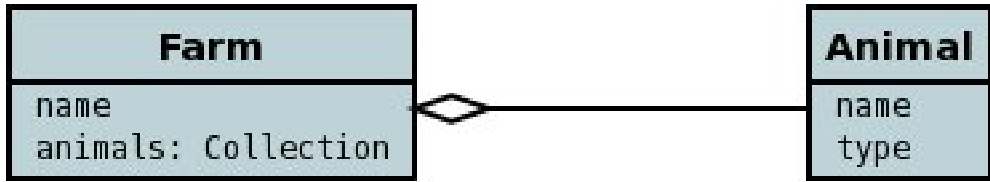
JPA Serialising Objects

JPA1 provides a way for users to specify that a field will be persisted *serialised*. This is of use, for example, to collections/maps/arrays which typically are stored using join tables or foreign-keys to other records. By specifying that a field is serialised a column will be added to store that field and the field will be serialised into it.

JPA1's definition of serialising applies to any field and all in the same way, unlike the situation with JDO2 which provides much more flexibility. Perhaps the most important thing to bear in mind when deciding to serialise a field is that that object in the field being serialised must implement *java.io.Serializable*.

Serialised Fields

If you wish to serialise a particular field into a single column (in the table of the class), you need to simply mark the field as a "lob" (large object). Let's take an example. We have the following classes



and we want the *animals* collection to be serialised into a single column in the table storing the Farm class, so we define our MetaData like this

```
<entity class="Farm">
    <table name="FARM"/>
    <attributes>
        ...
        <basic name="animals">
            <column name="ANIMALS"/>
            <lob/>
        </basic>
        ...
    </attributes>
</entity>
```

So we make use of the *lob* element (or @Lob annotation). This specification results in a table like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Provisos to bear in mind are

- Queries cannot be performed on collections stored as serialised.

If the field that we want to serialise is of type String, byte[], char[], Byte[] or Character[] then the field will be serialised into a CLOB column rather than BLOB.

See also :-

- MetaData reference for <basic> element
- Annotations reference for @Lob

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.6 Constraints

Constraints

A datastore often provides ways of constraining the storage of data to maintain relationships and improve performance. These are known as *constraints* and they come in various forms. These are :-

- Indexes - these are used to mark fields that are referenced often as indexes so that when they are used the performance is optimised.
- Unique constraints - these are placed on fields that should have a unique value. That is, only one object will have a particular value.
- Foreign-Keys - these are used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.
- Primary-Keys - allow the PK to be set, and also to have a name.

Unique constraints

Relational Databases (RDBMS) provide the ability to have unique constraints defined on tables to give extra control over data integrity. JPA1 provides a mechanism for defining such unique constraints. Let's take an example class, and show how to specify this

```
public class Person
{
    String forename;
    String surname;
    String nickname;
    ...
}
```

and here we want to impose uniqueness on the "nickname" field, so there is only one Person known as "DataNucleus Guru" for example !

```
<entity class="Person">
    <table name="PEOPLE"/>
    <attributes>
        ...
        <basic name="nickname">
            <column name="SURNAME" unique="true"/>
        </basic>
        ...
    </attributes>
</entity>
```

The second use of unique constraints is where we want to impose uniqueness across composite columns. So we reuse the class above, and this time we want to impose a constraint that there is only one Person with a particular "forename+surname".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<entity class="Person">
    <table name="PEOPLE">
        <unique-constraint>
            <column-name>FORENAME</column-name>
            <column-name>SURNAME</column-name>
        </unique-constraint>
    </table>
    <attributes>
        ...
        <basic name="forename">
            <column name="FORENAME"/>
        </basic>
        <basic name="surname">
            <column name="SURNAME"/>
        </basic>
        ...
    </attributes>
</entity>
```

In the same way we can also impose unique constraints on <join-table> and <secondary-table>

See also :-

- MetaData reference for <column> element
- MetaData reference for <unique-constraint> element
- Annotations reference for @Column
- Annotations reference for @UniqueConstraint

Indexes

The majority of datastores provide the ability to have indexes defined to give performance benefits. With RDBMS the indexes are specified on the table and the indexes to the rows are stored separately. In the same way an ODBMS typically allows indexes to be specified on the fields of the class, and these are managed by the datastore. JPA1 does not provide a way of defining indexes, but when using DataNucleus to create the datastore schema, will create indexes on all foreign key and primary key fields.

Foreign Keys

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe ? JPA1 doesnt allow the user to define any foreign-keys in the datastore. The only thing you can define is the cascading of delete operations, which is described elsewhere.

Primary Keys

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. You specify in other parts of the MetaData which fields are part of the primary key (if using application identity).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Unfortunately JPA1 doesnt allow specification of the name of the primary key constraint, nor of whether join tables are given a primary key constraint at all.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.7 Inheritance

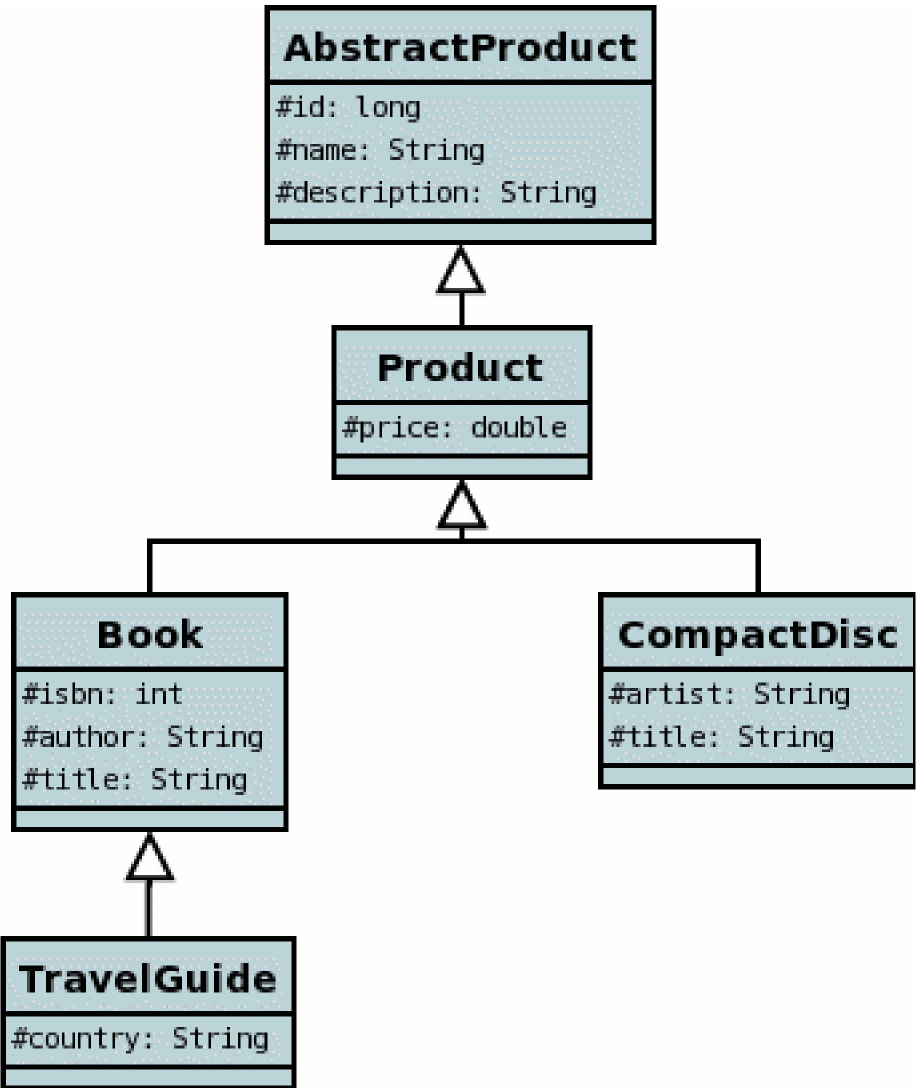
JPA Inheritance Strategies

In Java it is a normal situation to have inheritance between classes. With JPA you have choices to make as to how you want to persist your classes for the inheritance tree. For each inheritance tree (for the root class) you select how you want to persist those classes information. You have the following choices.

1. The *default strategy* is to select a class to have its fields persisted in the table of the base class. There is only one table per inheritance hierarchy. In JPA1 this is known as SINGLE_TABLE.
2. The next way is to have a table for each class in the inheritance hierarchy, and for each table to only hold columns for the fields of that class. Fields of superclasses are persisted into the table of the superclass. Consequently to get all field values for a subclass object a join is made of all tables of superclasses. In JPA1 this is referred to as JOINED
3. The third way is like JOINED except that each table will also contain columns for all inherited fields. In JPA1 this is referred to as TABLE_PER_CLASS.

In order to demonstrate the various inheritance strategies we need an example. Here are a few simple classes representing products in a (online) store. We have an abstract base class, extending this to to provide something that we can represent any product by. We then provide a few specialisations for typical products. We will use these classes later when defining how to persistent these objects in the different inheritance strategies.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The default JPA1 strategy is "SINGLE_TABLE", namely that the base class will have a table and all subclasses will be persisted into that same table. So if you dont specify an "inheritance strategy" in your root class this is what you will get.

See also :-

- MetaData reference for <inheritance> element
- MetaData reference for <discriminator-column> element
- Annotations reference for @Inheritance
- Annotations reference for @DiscriminatorColumn

SINGLE_TABLE

"SINGLE_TABLE" strategy is where the root class has a table and all subclasses are also persisted into that table. This corresponds to JDO2s "new-table" for the root class and "superclass-table" for all subclasses. This has the advantage that retrieval of an object is a single SQL call to a single table. It also has the disadvantage that the single table can have a very large number of columns, and database

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

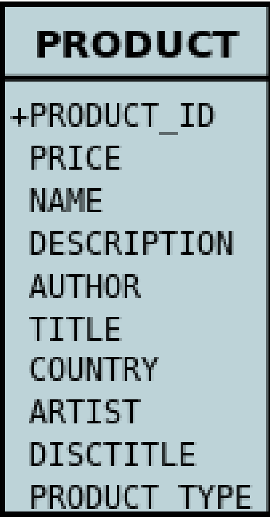
readability and performance can suffer, and additionally that a discriminator column is required. In our example, lets ignore the AbstractProduct class for a moment and assume that Product is the base class (with the "id"). We have no real interest in having separate tables for the Book and CompactDisc classes and want everything stored in a single table PRODUCT. We change our MetaData as follows

```
<entity name="Product">
    <inheritance strategy="SINGLE_TABLE"/>
    <discriminator-value>PRODUCT</discriminator-value>
    <discriminator-column name="PRODUCT_TYPE" discriminator-type="STRING"/>
    <attributes>
        <id name="id">
            <column name="PRODUCT_ID"/>
        </id>
        <basic name="price">
            <column name="PRICE"/>
        </basic>
    </attributes>
</entity>
<entity name="Book">
    <discriminator-value>BOOK</discriminator-value>
    <attributes>
        <basic name="isbn">
            <column name="ISBN"/>
        </basic>
        <basic name="author">
            <column name="AUTHOR"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
<entity name="TravelGuide">
    <discriminator-value>TRAVELGUIDE</discriminator-value>
    <attributes>
        <basic name="country">
            <column name="COUNTRY"/>
        </basic>
    </attributes>
</entity>
<entity name="CompactDisc">
    <discriminator-value>COMPACTDISC</discriminator-value>
    <attributes>
        <basic name="artist">
            <column name="ARTIST"/>
        </basic>
        <basic name="title">
            <column name="DISCTITLE"/>
        </basic>
    </attributes>
</entity>
```

This change of use of the inheritance element has the effect of using the PRODUCT table for all classes, containing the fields of Product, Book, CompactDisc, and TravelGuide. You will also note that we used a discriminator-column element for the Product class. The specification above will result in an extra column (called PRODUCT_TYPE) being added to the PRODUCT table, and containing the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

"discriminator-value" of the object stored. So for a Book it will have "BOOK" in that column for example. This column is used in discriminating which row in the database is of which type. The final thing to note is that in our classes Book and CompactDisc we have a field that is identically named. With CompactDisc we have defined that its column will be called DISCTITLE since both of these fields will be persisted into the same table and would have had identical names otherwise - this gets around the problem.



In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into the PRODUCT table only.

JOINED

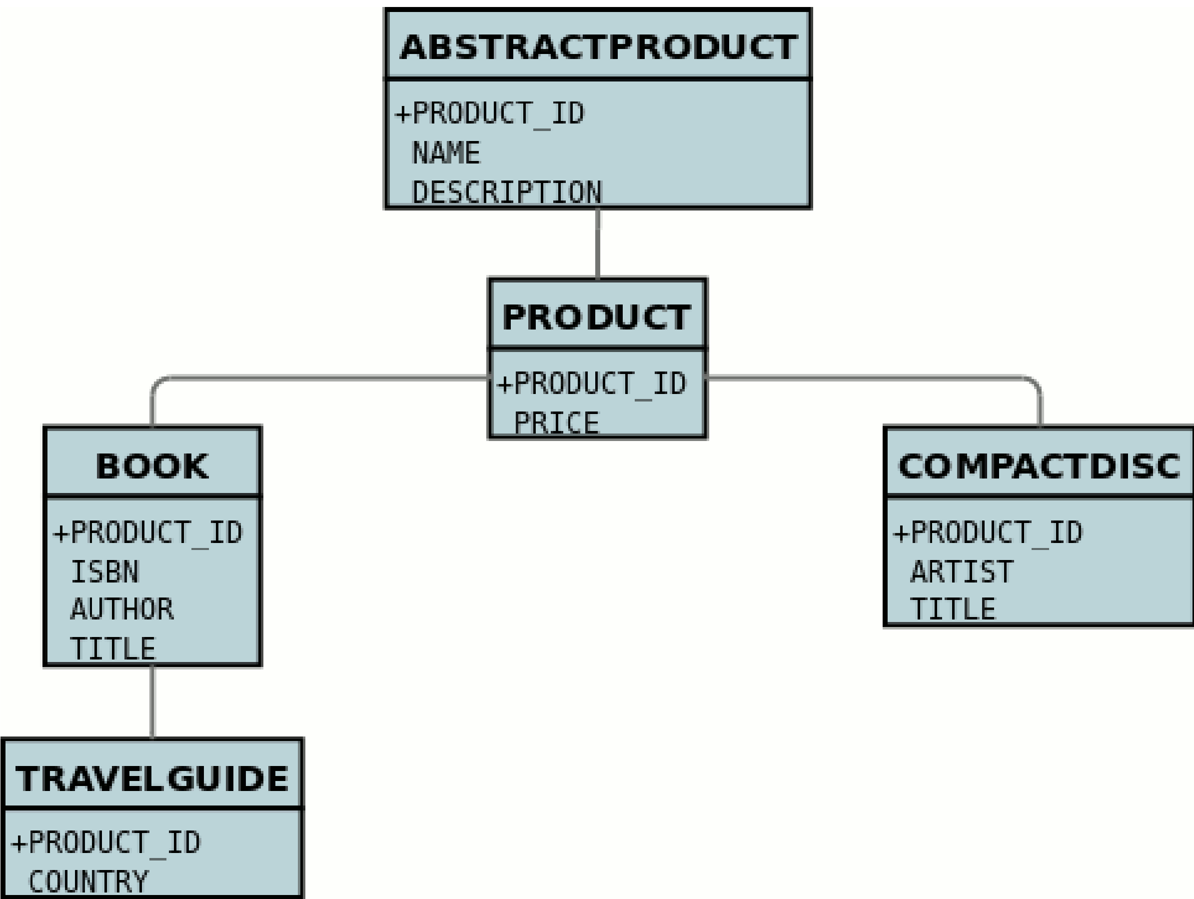
"JOINED" strategy means that each table in the inheritance hierarchy has its own table and that the table of each class only contains columns for that class. Inherited fields are persisted into the tables of the superclass(es). This corresponds to JDO2s "new-table" (for all classes in the inheritance hierarchy). This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub-type. Let's try an example using the simplest to understand strategy JOINED. We have the classes defined above, and we want to persist our classes each in their own table. We define the Meta-Data for our classes like this

```
<entity class="AbstractProduct">
    <inheritance strategy="JOINED"/>
    <attributes>
        <id name="id">
            <column name="PRODUCT_ID"/>
        </id>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="description">
            <column name="DESCRIPTION"/>
        </basic>
    </attributes>
</entity>
<entity class="Product">
```

```
    <attributes>
        <basic name="price">
            <column name="PRICE"/>
        </basic>
    </attributes>
</entity>
<entity class="Book">
    <attributes>
        <basic name="isbn">
            <column name="ISBN"/>
        </basic>
        <basic name="author">
            <column name="AUTHOR"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
<entity class="TravelGuide">
    <attributes>
        <basic name="country">
            <column name="COUNTRY"/>
        </basic>
    </attributes>
</entity>
<entity class="CompactDisc">
    <attributes>
        <basic name="artist">
            <column name="ARTIST"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
```

So we will have 5 tables - ABSTRACTPRODUCT, PRODUCT, BOOK, COMPACTDISC, and TRAVELGUIDE. They each contain just the fields for that class (and not any inherited fields, except the identity to join with).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into ABSTRACTPRODUCT, PRODUCT, BOOK, and TRAVELGUIDE.

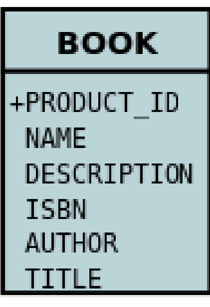
TABLE_PER_CLASS

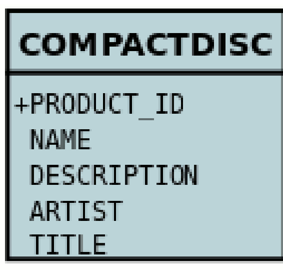
This strategy is like "JOINED" except that in addition to each class having its own table, the table also holds columns for all inherited fields. So taking the same classes as used above

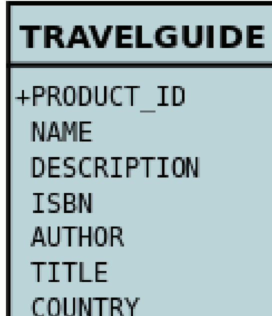
```
<entity class="AbstractProduct">
    <inheritance strategy="TABLE_PER_CLASS"/>
    <attributes>
        <id name="id">
            <column name="PRODUCT_ID"/>
        </id>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="description">
            <column name="DESCRIPTION"/>
        </basic>
    </attributes>
</entity>
<entity class="Product">
    <attributes>
        <basic name="price">
            <column name="PRICE"/>
```

```
        </basic>
    </attributes>
</entity>
<entity class="Book">
    <attributes>
        <basic name="isbn">
            <column name="ISBN"/>
        </basic>
        <basic name="author">
            <column name="AUTHOR"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
<entity class="TravelGuide">
    <attributes>
        <basic name="country">
            <column name="COUNTRY"/>
        </basic>
    </attributes>
</entity>
<entity class="CompactDisc">
    <attributes>
        <basic name="artist">
            <column name="ARTIST"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
```

This then implies a datastore schema as follows







So any object of explicit type Book is persisted into the table BOOK. Similarly any TravelGuide is persisted into the table TRAVELGUIDE, etc In addition if any class in the inheritance tree is abstract then it won't have a table since there cannot be any instances of that type. **DataNucleus currently has limitations when using a class using this inheritance as the element of a collection.**

Mapped Superclasses

JPA defines entities called "mapped superclasses" for the situation where you dont persist an actual object of a superclass type but that all subclasses of that type that are entities will also persist the values

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

for the fields of the "mapped superclass". That is a "mapped superclass" has no table to store its objects in a datastore. Instead its fields are stored in the tables of its subclasses. Let's take an example

```
<mapped-superclass class="AbstractProduct">
    <attributes>
        <id name="id">
            <column name="PRODUCT_ID"/>
        </id>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="description">
            <column name="DESCRIPTION"/>
        </basic>
    </attributes>
</mapped-superclass>

<entity class="Product">
    <attributes>
        <basic name="price">
            <column name="PRICE"/>
        </basic>
    </attributes>
</entity>
```

In this case we will have a table for Product and the fields of AbstractProduct will be stored in this table. If the mapping information (column names etc) for these fields need setting then you should use <attribute-override> in the MetaData for Product.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.8 Arrays

JPA Arrays

JPA defines support the persistence of arrays but only arrays of byte[], Byte[], char[], Character[]. Moreover it only defines support for persisting these arrays into CLOB columns, hence they have to be byte-streamed. Namely

- Single Column - the array is byte-streamed into a single column in the table of the containing object.

Single Column Arrays

Let's suppose you have a class something like this

Account
firstName: String
lastName: String
permissions: byte[]

So we have an Account and it has a number of permissions, each expressed as a byte. We want to persist the permissions in a single-column into the table of the account (but we don't want them serialised). We then define MetaData something like this

```
<entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
        ...
        <basic name="permissions">
            <column name="PERMISSIONS"/>
            <lob/>
        </basic>
        ...
    </attributes>
</entity>
```

This results in a datastore schema as follows

See also :-

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- MetaData reference for <basic> element
- Annotations reference for @Basic

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7.9 Versioning

JPA Versioning

JPA1 allows objects of classes to be versioned. The version is typically used as a way of detecting if the object has been updated by another thread or EntityManager since retrieval using the current EntityManager - for use by Optimistic Transactions. JPA1s mechanism for versioning of objects in RDBMS datastores is to mark a field of the class to store the version. The field must be Integer/Long based.

With JPA1 you can specify the details of this **version field** as follows.

```
<entity name="mydomain.User">
    <attributes>
        <id name="id"/>
        <version name="version"/>
    </attributes>
</entity>
```

or alternatively using annotations

```
@Entity
public class User
{
    @Id
    long id;

    @Version
    int version;

    ...
}
```

The specification above will use the "version" field for storing the version of the object. DataNucleus will use a "version-number" strategy for populating the value (see JDO2 Versioning for details of the strategies that JDO2 allows).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.1 Managing Relationships

Relationships

The power of a Java persistence solution like DataNucleus is demonstrated when persisting relationships between objects. There are many types of relationships.

- 1-1 relationships - this is where you have an object A relates to a second object B. The relation can be *unidirectional* where A knows about B, but B doesnt know about A. The relation can be *bidirectional* where A knows about B and B knows about A.
- 1-N relationships - this is where you have an object A that has a collection of other objects of type B. The relation can be *unidirectional* where A knows about the objects B but the Bs dont know about A. The relation can be *bidirectional* where A knows about the objects B and the Bs know about A
- N-1 relationships - this is where you have an object B1 that relates to an object A, and an object B2 that relates to A also etc. The relation can be *unidirectional* where the A doesnt know about the Bs. The relation can be *bidirectional* where the A has a collection of the Bs. [i.e a 1-N relationship but from the point of view of the element]
- M-N relationships - this is where you have objects of type A that have a collection of objects of type B and the objects of type B also have a collection of objects of type A. The relation is always *bidirectional* by definition

Assigning Relationships

When the relation is *unidirectional* you simply set the related field to refer to the other object. For example we have classes A and B and the class A has a field of type B. So we set it like this

```
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
```

When the relation is *bidirectional* you **have to set both sides** of the relation. For example, we have classes A and B and the class A has a collection of elements of type B, and B has a field of type A. So we set it like this

```
A a = new A();
B b1 = new B();
a.addElement(b1); // "a" knows about "b1"
b1.setA(a); // "b1" knows about "a"
```

So it is really simple, with only 1 real rule. **With a *bidirectional* relation you must set both sides of the relation**

Persisting Relationships - Reachability

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

To persist an object with JPA you call the *EntityManager* method *persist*. The object passed in will be persisted. By default all related objects will **not** be persisted with that object. You can however change this by specifying the cascade PERSIST property for that field. With this the related object(s) would also be persisted (or updated with any new values if they are already persistent). This process is called **persistence-by-reachability**. With JDO the default is to persist all reachables, whereas with JPA you have to explicitly set this behaviour. For example we have classes A and B and class A has a field of type B and this field has the cascade property PERSIST set. To persist them we could do

```
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
em.persist(a);
```

Managed Relationships

As we have mentioned above, it is for the user to set both sides of a bidirectional relation. If they dont and object A knows about B, but B doesnt know about A then what is the persistence solution to do ? It doesnt know which side of the relation is correct. JPA doesnt define the behaviour currently for this situation. DataNucleus has two ways of handling this situation. If you have the persistence property "datanucleus.manageRelationships" set to true then it will make sure that the other side of the relation is set correctly, correcting obvious omissions, and giving exceptions for obvious errors. If you set that persistence property to false then it will assume that your objects have their bidirectional relationships consistent and will just persist what it finds.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.2 Cascading

Cascading Fields

When defining your objects to be persisted and the relationships between them, it is often required to define dependencies between these related objects. When persisting an object should we also persist any related objects? What should happen to a related object when an object is deleted ? Can the related object exist in its own right beyond the lifecycle of the other object, or should it be deleted along with the other object ? This behaviour can be defined with JPA1 and with DataNucleus. Lets take an example

```
@Entity
public class Owner
{
    @OneToOne
    private DrivingLicense license;

    @OneToMany(mappedBy="owner")
    private Collection cars;

    ...
}

@Entity
public class DrivingLicense
{
    private String serialNumber;

    ...
}

@Entity
public class Car
{
    private String registrationNumber;

    @ManyToOne
    private Owner owner;

    ...
}
```

So we have an Owner of a collection of vintage Car's, and the Owner has a DrivingLicense. We want to define lifecycle dependencies to match the relationships that we have between these objects. So in our example what we are going to do is

- When an object is persisted/updated its related objects are also persisted/updated.
- When an Owner object is deleted, its DrivingLicense is deleted too (since it cant exist without the person!
- When an Owner object is deleted, the Cars continue to exist (since someone will buy them)
- When a Car object is deleted, the Owner continues to exist (unless he/she dies in the Car, but that

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

will be handled by a different mechanism in our application).

So we update our class to reflect this

```
@Entity
public class Owner
{
    @OneToOne(cascade=CascadeType.ALL)
    private DrivingLicense license;

    @OneToMany(mappedBy="owner", cascade={CascadeType.PERSIST, CascadeType.MERGE})
    private Collection cars;

    ...
}

@Entity
public class DrivingLicense
{
    private String serialNumber;

    ...
}

@Entity
public class Car
{
    private String registrationNumber;

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
    private Owner owner;

    ...
}
```

So we make use of the *cascade* attribute of the relation annotations. We could express this similarly in XML

```
<entity-mappings>
    <entity class="mydomain.Owner">
        <attributes>
            <one-to-many name="cars">
                <cascade>
                    <cascade-persist/>
                    <cascade-merge/>
                </cascade>
            </one-to-many>
            <one-to-one name="license">
                <cascade>
                    <cascade-all/>
                </cascade>
            </one-to-one>
            ...
        </attributes>
    </entity>

    <entity class="mydomain.DrivingLicense">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        ...
    </entity>

    <entity class="mydomain.Car">
        <attributes>
            <many-to-one name="owner">
                <cascade>
                    <cascade-persist/>
                    <cascade-merge/>
                </cascade>
            </many-to-one>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.3 1-to-1

JPA 1-1 Relationships

You have a 1-to-1 relationship when an object of a class has an associated object of another class (only one associated object). It could also be between an object of a class and another object of the same class (obviously). You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only one of the classes knows about the other class (unidirectional). These are described below.

Unidirectional

For this case you could have 2 classes, User and Account, as below.
so the Account class knows about the User class, but not vice-versa. If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="User">
        <table name="USER"/>
        <attributes>
            <id name="id">
                <column name="USER_ID"/>
            </id>
            <basic name="login">
                <column name="LOGIN" length="20"/>
            </basic>
    </entity>

    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
            <one-to-one name="user">
                <join-column name="USER_ID"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

This will create 2 tables in the database, one for User (with name USER), and one for Account (with name ACCOUNT and a column USER_ID), as shown below.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Things to note :-

- Account has the object reference to User (and so is the "owner" of the relation) and so its table holds the foreign-key
- If you call EntityManager.remove() on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.

Bidirectional

For this case you could have 2 classes, User and Account again, but this time as below. Here the Account class knows about the User class, and also vice-versa.

We create the 1-1 relationship with a single foreign-key. To do this you define the MetaData as

```
<entity-mappings>
    <entity class="User">
        <table name="USER"/>
        <attributes>
            <id name="id">
                <column name="USER_ID"/>
            </id>
            <basic name="login">
                <column name="LOGIN" length="20"/>
            </basic>
            <one-to-one name="account" mapped-by="user"/>
        </attributes>
    </entity>

    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
            <one-to-one name="user">
                <join-column name="USER_ID"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The difference is that we added mapped-by to the field of User. This will create 2 tables in the database, one for User (with name USER), and one for Account (with name ACCOUNT including a USER_ID). The fact that we specified the mapped-by on the User class means that the foreign-key is created in the ACCOUNT table.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Things to note :-

- The "mapped-by" is specified on User (the non-owning side) and so the foreign-key is held by the table of Account (the owner of the relation)
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4 1-to-N

JPA 1-N Relationships

You have a 1-N (one to many) when you have one object of a class that has a Collection/Map of objects of another class. In the *java.util* package there are an assortment of possible collection/map classes and they all have subtly different behaviour with respect to allowing nulls, allowing duplicates, providing ordering, etc. There are two ways in which you can represent a collection or map in a datastore : Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the collection or map.

We split our documentation based on what type of collection/map you are using.

- 1-N using Collection types
- 1-N using Set types
- 1-N using List type
- 1-N using Map type

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4.1 1-to-N (Collection)

JPA 1-N Relationships with Collections

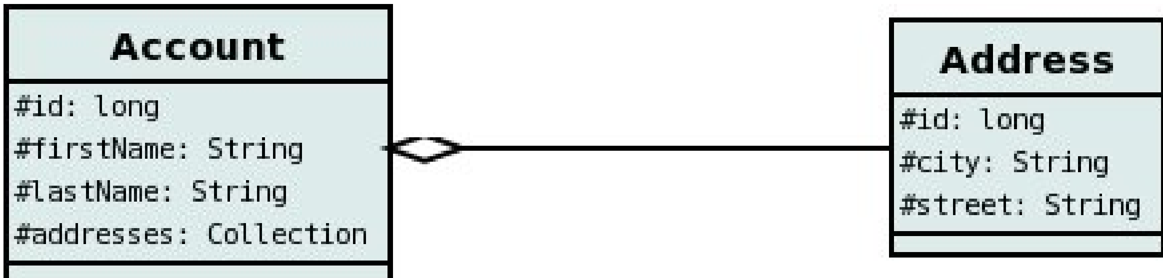
You have a 1-N (one to many) when you have one object of a class that has a Collection of objects of another class. Please note that Collections allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore : Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Collection.

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table

1-N Collection Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join-table element on the field element - this signals to JPA to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the collection.
- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will NOT be given a primary key (since a Collection can have duplicates).

Using Foreign-Key

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

In strict JPA1 you cannot have a 1-N unidirectional relation using a ForeignKey - they must have a JoinTable. Consequently to use this relation as specified below you must specify the property "datanucleus.jpa.level" as "JPA2"/"DataNucleus"

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

There will be 2 tables, one for Address, and one for Account. If you wish to specify the names of the column(s) used in the schema for the foreign key in the Address table you should use the join-column element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

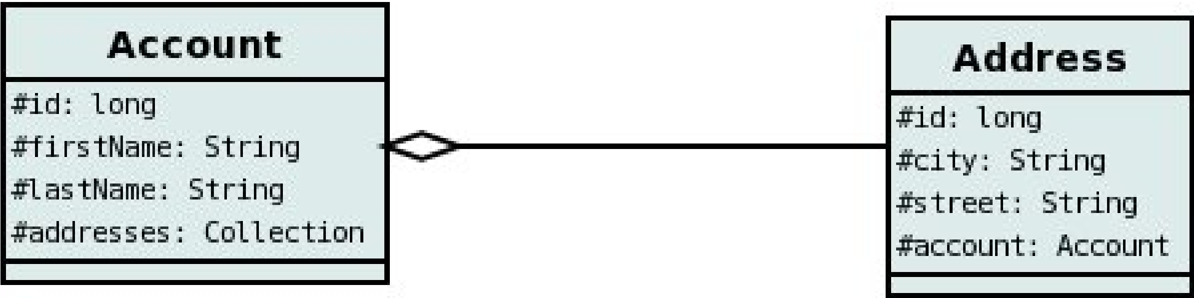
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

1-N Collection Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join element on the field element - this signals to JPA to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the collection.
- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will NOT be given a primary key (since a Collection can have duplicates).
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called account on the Address class. This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the classelement
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

1-N Collection of non-Entity objects

In strict JPA1 you cannot have a 1-N collection of non-Entity objects. To use this relation as

ALL RIGHTS RESERVED

specified below you must specify the property "datanucleus.jpa.level" as "JPA2"/"DataNucleus" All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this

```
@Entity
public class Account
{
    ...

    @ElementCollection(targetClass=String.class)
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection addresses;
}
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". How to define the column name of the element in the join table is currently undefined in JPA2.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4.2 1-to-N (Set)

JPA 1-N Relationships with Sets

You have a 1-N (one to many) when you have one object of a class that has a Set of objects of another class. Please note that Sets do not allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore : Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Set.

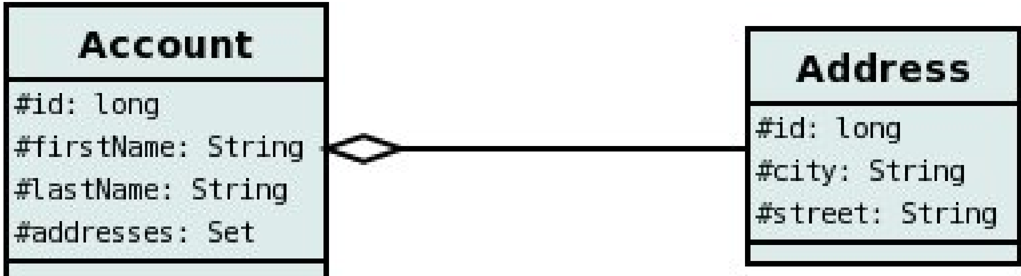
The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table

This page is aimed at Set fields and so applies to fields of Java type *java.util.HashSet, java.util.LinkedHashSet, java.util.Set, java.util.SortedSet, java.util.TreeSet*

1-N Set Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join-table element on the field element - this signals to JPA to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

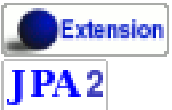
The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the Set
- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will be given a primary key (since a Set can't have duplicates).

Using Foreign-Key

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



In strict JPA1 you cannot have a 1-N unidirectional relation using a ForeignKey - they must have a JoinTable. Consequently to use this relation as specified below you must specify the property "datanucleus.jpa.level" as "JPA2"/"DataNucleus"

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

There will be 2 tables, one for Address, and one for Account. If you wish to specify the names of the column(s) used in the schema for the foreign key in the Address table you should use the join-column element within the field of the Set.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account Set field since the Address knows nothing about the Account.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

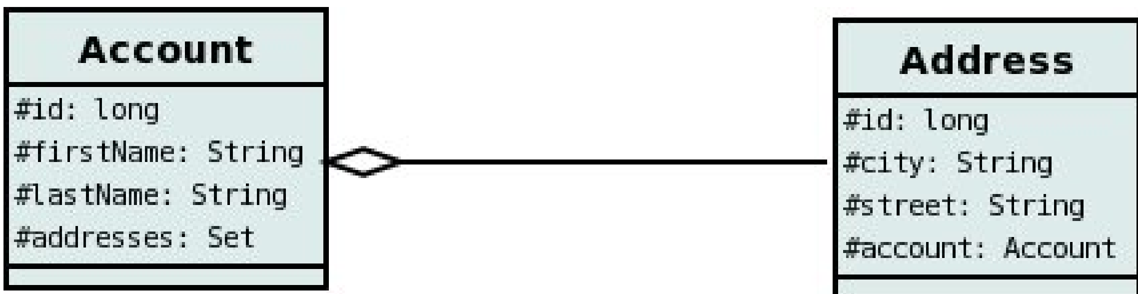
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Set. If you want to allow duplicate Set entries, then use the "Join Table" variant above.

1-N Set Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join element on the field element - this signals to JPA to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the set.
- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will be given a primary key (since a Set can't have duplicates).
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called account on the Address class. This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the classelement
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Set. If you want to allow duplicate Set entries, then use the "Join Table" variant above.

1-N Collection of non-Entity objects

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

In strict JPA1 you cannot have a 1-N collection of non-Entity objects. To use this relation as specified below you must specify the property "datanucleus.jpa.level" as "JPA2"/"DataNucleus" All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this

```
@Entity
public class Account
{
    ...

    @ElementCollection(targetClass=String.class)
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection addresses;
}
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". How to define the column name of the element in the join table is currently undefined in JPA2.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4.3 1-to-N (List)

JPA 1-N/N-1 Relationships with Lists

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a List of objects of another class. There are two ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the List.

The various possible relationships are described below.

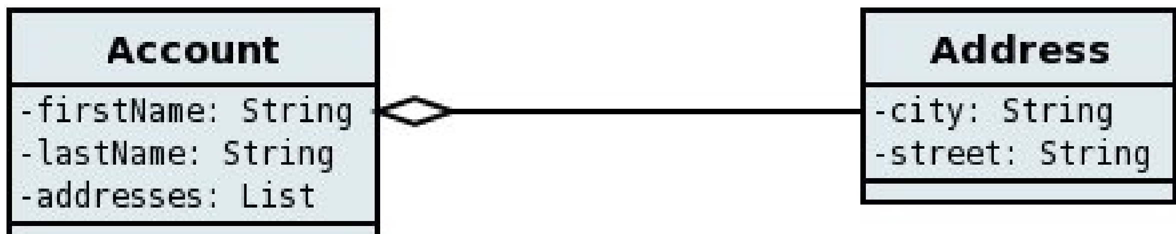
- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table

This page is aimed at List fields and so applies to fields of Java type *java.util.ArrayList, java.util.LinkedList, java.util.List, java.util.Stack, java.util.Vector*

In JPA1 all List relationships are "ordered Lists". If a List is an "ordered List" then the positions of the elements in the List at persistence are not preserved (are not persisted) and instead an ordering is defined for their retrieval. **In JPA2 Lists can optionally use a surrogate column to handle the ordering**. This means that the positions of the elements in List at persistence are preserved. This is the same situation as JDO provides.

1-N List Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <order-by>id</order-by>
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join-table element on the field element - this signals to JPA to use a join table. There will be 3 tables, one for Address, one for Account, and the join table. This is identical to the handling for Sets/Collections. Note that we specified <order-by> which defines the order the elements are retrieved in (the "id" is the field in the List element).

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

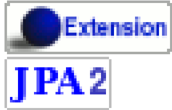
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the collection.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will NOT be given a primary key (so that duplicates can be stored).
- If you want to have a surrogate column added to contain the ordering you should specify order-column (@OrderColumn) instead of order-by. This is available from JPA2

Using Foreign-Key



In strict JPA1 you cannot have a 1-N unidirectional relation using a ForeignKey - they must have a JoinTable. Consequently to use this relation as specified below you must specify the property "datanucleus.jpa.level" as "JPA2"/"DataNucleus"

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <order-by>city</order-by>
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

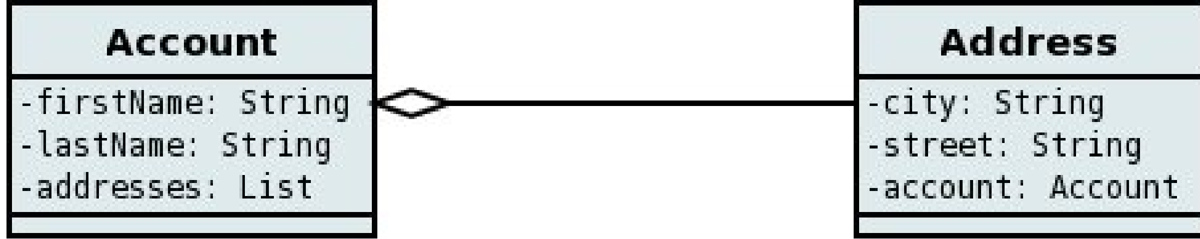
- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.

Limitations

- If we are using an "ordered List" and the primary key of the join table contains owner and element then duplicate elements can't be stored in a List. If you want to allow duplicate List entries, then use the "Join Table" variant above.

1-N List Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this



There are 2 ways that we can persist this relationship. These are shown below

Using Join Table

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <order-by>id</order-by>
                <join-table name="ACCOUNT_ADDRESSES">
                    <join-column name="ACCOUNT_ID_OID"/>
                    <inverse-join-column name="ADDRESS_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the join element on the field element - this signals to JPA to use a join table. This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the collection.
- To specify the names of the join table columns, use the join-column and *inverse-join-column* elements below the join-table element.
- The join table will NOT be given a primary key (so that duplicates can be stored).
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want to have a surrogate column added to contain the ordering you should specify order-column (@OrderColumn) instead of order-by. This is available from JPA2

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

Please note that an Foreign-Key List will NOT, by default, allow duplicates. This is because it stores the element position in the element table. If you need a List with duplicates we recommend that you use the Join Table List implementation above. If you have an application identity element class then you could in principle add the element position to the primary key to allow duplicates, but this would imply changing your element class identity.

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <order-by>city ASC</order-by>
                <join-column name="ACCOUNT_ID"/>
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <many-to-one name="account">
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called account on the Address class. This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the classelement
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation : Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

1-N Collection of non-Entity objects

In strict JPA1 you cannot have a 1-N collection of non-Entity objects. To use this relation as specified below you must specify the property *datanucleus.jpa.level* as "JPA2"/"DataNucleus"
All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this

```
@Entity
public class Account
{
    ...

    @ElementCollection(targetClass=String.class)
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection addresses;
}
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". How to define the column name of the element in the join table is currently undefined in JPA2.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4.4 1-to-N (Map)

JPA 1-N/N-1 Relationships with Maps

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Map of objects of another class. There are two general ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Map. JPA only supports the Foreign-Key approach where you have the key stored as a field in the value.

The various possible relationships are described below.

- 1-N Unidirectional using Foreign-Key (key stored in the value class)
- 1-N Bidirectional using Foreign-Key (key stored in the value class)

This page is aimed at Map fields and so applies to fields of Java type *java.util.HashMap, java.util.Hashtable, java.util.LinkedHashMap, java.util.Map, java.util.SortedMap, java.util.TreeMap, java.util.Properties*

1-N Map using Foreign-Key

1-N Foreign-Key Unidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value. We're using a field (*alias*) in the Address class as the key of the map.

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address">
                <map-key name="alias"/>
                <join-column name="ACCOUNT_ID_OID"/>
            </one-to-many>
        </attributes>
    </entity>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <basic name="alias">
                <column name="KEY" length="20"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Again there will be 2 tables, one for Address, and one for Account. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the join-column element within the field of the map.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account.

1-N Foreign-Key Bidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value.

With these classes we want to store a foreign-key in the value table (ADDRESS), and we want to use the "alias" field in the Address class as the key to the map. If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="100"/>
            </basic>
            <basic name="lastName">
                <column name="LASTNAME" length="100"/>
            </basic>
            <one-to-many name="addresses" target-entity="com.mydomain.Address"
mapped-by="account">
                <map-key name="alias"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            </one-to-many>
        </attributes>
    </entity>

    <entity class="Address">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <column name="ADDRESS_ID"/>
            </id>
            <basic name="city">
                <column name="CITY" length="50"/>
            </basic>
            <basic name="street">
                <column name="STREET" length="50"/>
            </basic>
            <basic name="alias">
                <column name="KEY" length="20"/>
            </basic>
            <many-to-one name="account">
                <join-column name="ACCOUNT_ID_OID"/>
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

This will create 2 tables in the datastore. One for Account, and one for Address. The table for Address will contain the key field as well as an index to the Account record (notated by the mapped-by tag).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.5 N-to-1

JPA N-1 Relationships

You have a N-to-1 relationship when an object of a class has an associated object of another class (only one associated object) and several of this type of object can be linked to the same associated object. From the other end of the relationship it is effectively a 1-N, but from the point of view of the object in question, it is N-1. You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only the "N" side knows about the other class (unidirectional). These are described below.

Unidirectional

For this case you could have 2 classes, User and Account, as below.
so the Account class ("N" side) knows about the User class ("1" side), but not vice-versa. A particular user could be related to several accounts. If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="User">
        <table name="USER"/>
        <attributes>
            <id name="id">
                <column name="USER_ID"/>
            </id>
            <basic name="login">
                <column name="LOGIN" length="20"/>
            </basic>
        </attributes>
    </entity>

    <entity class="Account">
        <table name="ACCOUNT"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
            <many-to-one name="user">
                <join-column name="USER_ID"/>
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

This will create 2 tables in the database, one for User (with name USER), and one for Account (with

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

name ACCOUNT and a column USER_ID), as shown below.

Things to note :-

- If you wish to specify the names of the database tables and columns for these classes, you can use the element table (under the entity element) and the attribute name (on the column element)
- If you call PM.deletePersistent() on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.

Bidirectional

This relationship is described in the guide for 1-N relationships. In particular there are 2 ways to define the relationship. The first uses a Join Table to hold the relationship. The second uses a Foreign Key in the "N" object to hold the relationship. Please refer to the 1-N relationships bidirectional relations since they show this exact relationship.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.6 M-to-N

JPA M-N Relationships

You have a M-to-N (or Many-to-Many) relationship if an object of a class A has associated objects of class B, and class B has associated objects of class A. This relationship may be achieved through Java Collection, Set, List or subclasses of these, although the only one that supports a true M-N is Set.

With DataNucleus this can be set up as described in this section, using what is called a Join Table relationship. Let's take the following example and describe how to model it with the different types of collection classes. We have 2 classes, Product and Supplier as below.

Here the Product class knows about the Supplier class. In addition the Supplier knows about the Product class, however with these relationships are really independent.

Using Set

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="mydomain.Product">
        <table name="PRODUCT"/>
        <attributes>
            <id name="id">
                <column name="PRODUCT_ID"/>
            </id>
            <basic name="name">
                <column name="NAME"/>
            </basic>
            <basic name="price">
                <column name="PRICE"/>
            </basic>
            <many-to-many name="suppliers" mapped-by="products">
                <join-table name="PRODUCTS_SUPPLIERS">
                    <join-column name="PRODUCT_ID"/>
                    <inverse-join-column name="SUPPLIER_ID"/>
                </join-table>
            </many-to-many>
        </attributes>
    </entity>

    <entity class="mydomain.Supplier">
        <table name="SUPPLIER"/>
        <attributes>
            <id name="id">
                <column name="SUPPLIER_ID"/>
            </id>
            <basic name="name">
                <column name="NAME"/>
            </basic>
            <many-to-many name="products"/>
        </attributes>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </entity>
</entity-mappings>
```

Note how we have specified the information only once regarding join table name, and join column names as well as the <join-table>. This is the JPA standard way of specification, and results in a single join table. The "mapped-by" ties the two fields together.

See also :-

- M-N Worked Example
- M-N with Attributes Worked Example

Using List

If you define the Meta-Data for these classes as follows

```
<entity-mappings>
    <entity class="mydomain.Product">
        <table name="PRODUCT"/>
        <attributes>
            <id name="id">
                <column name="PRODUCT_ID"/>
            </id>
            <basic name="name">
                <column name="NAME"/>
            </basic>
            <basic name="price">
                <column name="PRICE"/>
            </basic>
            <many-to-many name="suppliers" mapped-by="products">
                <order-by>name</order-by>
                <join-table name="PRODUCTS_SUPPLIERS">
                    <join-column name="PRODUCT_ID"/>
                    <inverse-join-column name="SUPPLIER_ID"/>
                </join-table>
            </many-to-many>
        </attributes>
    </entity>

    <entity class="mydomain.Supplier">
        <table name="SUPPLIER"/>
        <attributes>
            <id name="id">
                <column name="SUPPLIER_ID"/>
            </id>
            <basic name="name">
                <column name="NAME"/>
            </basic>
            <many-to-many name="products">
                <order-by>name</order-by>
            </many-to-many>
        </attributes>
    </entity>
</entity-mappings>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

There will be 3 tables, one for Product, one for Supplier, and the join table. The difference from the Set example is that we now have <order-by> at both sides of the relation. This has no effect in the datastore schema but when the Lists are retrieved they are ordered using the specified order-by.

Relationship Behaviour

Please be aware of the following.

- To add an object to an M-N relationship you need to set it at both ends of the relation since the relation is bidirectional and without such information the JPA implementation won't know which end of the relation is correct.
- If you want to delete an object from one end of a M-N relationship you will have to remove it first from the other objects relationship. If you don't you will get an error message that the object to be deleted has links to other objects and so cannot be deleted.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.7 Compound Identity Relation

JPA Compound Identity Relationships

An identifying relationship (or "compound identity relationship" in JDO) is a relationship between two objects of two classes in which the child object must coexist with the parent object and where the primary key of the child includes the Entity object of the parent. So effectively the key aspect of this type of relationship is that the primary key of one of the classes includes a Entity field (hence why is is referred to as Compound Identity). **Please note that this is not part of the JPA1 specification and is allowed only by DataNucleus. Avoid this relation type if you want to keep your application implementation-independent.** This type of relation is available in the following forms

- 1-1 unidirectional
- 1-N collection bidirectional using ForeignKey
- 1-N map bidirectional using ForeignKey (key stored in value)

1-1 Relationship

Lets take the same classes as we have in the 1-1 Relationships. In the 1-1 relationships guide we note that in the datastore representation of the **User** and **Account** the ACCOUNT table has a primary key as well as a foreign-key to USER. In our example here we want to just have a primary key that is also a foreign-key to USER. To do this we need to modify the classes slightly and add primary-key fields and use "application-identity".

In addition we need to define primary key classes for our User and Account classes

```
public class User
{
    long id;

    ... (remainder of User class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Account
{
    User user;

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public User.PK user; // Use same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");

            this.user = new User.PK(token.nextToken());
        }

        public String toString()
        {
            return "" + this.user.toString();
        }

        public int hashCode()
        {
            return user.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.user.equals(otherPK.user);
            }
            return false;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
    <entity class="mydomain.User">
        <table name="USER"/>
        <id-class class="mydomain.User.PK"/>
        <attributes>
            <id name="id">
                <column name="USER_ID"/>
            </id>
            <basic name="login">
                <column name="LOGIN" length="20"/>
            </basic>
    </entity>

    <entity class="mydomain.Account">
        <table name="ACCOUNT"/>
        <id-class class="mydomain.Account.PK"/>
        <attributes>
            <id name="user">
                <column name="USER_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
            <one-to-one name="user"/>
        </attributes>
    </entity>
</entity-mappings>
```

So now we have the following datastore schema

Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- You can only have one "Account" object linked to a particular "User" object since the FK to the "User" is now the primary key of "Account". To remove this restriction you could also add a "long id" to "Account" and make the "Account.PK" a composite primary-key

1-N Collection Relationship

Lets take the same classes as we have in the 1-N Relationships (FK). In the 1-N relationships guide we

note that in the datastore representation of the **Account** and **Address** classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to *include* the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes

```
public class Account
{
    long id; // PK field

    Set addresses = new HashSet();

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Address
{
    long id;
    Account account;

    .. (remainder of Address class)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id; // Same name as real field above
        public Account.PK account; // Same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");
            this.id = Long.valueOf(token.nextToken()).longValue();
            this.account = new Account.PK(token.nextToken());
        }

        public String toString()
        {
            return "" + id + "::" + this.account.toString();
        }

        public int hashCode()
        {
            return (int)id ^ account.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id &&
this.account.equals(otherPK.account);
            }
            return false;
        }
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
    <entity class="mydomain.Account">
        <table name="ACCOUNT"/>
        <id-class class="mydomain.Account.PK"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <one-to-many name="addresses" mapped-by="account"/>
    </entity>

    <entity class="mydomain.Address">
        <table name="ADDRESS"/>
        <id-class class="mydomain.Address.PK"/>
        <attributes>
            <id name="id">
                <column name="ID"/>
            </id>
            <id name="account">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="city">
                <column name="CITY"/>
            </basic>
            <basic name="street">
                <column name="STREET"/>
            </basic>
            <many-to-one name="account"/>
        </attributes>
    </entity>
</entity-mappings>
```

So now we have the following datastore schema

Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- If we had omitted the "id" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "id" field too.

1-N Map Relationship

Lets take the same classes as we have in the 1-N Relationships (FK). In this guide we note that in the datastore representation of the **Account** and **Address** classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to *include* the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes

```
public class Account
{
    long id; // PK field

    Set addresses = new HashSet();

    ... (remainder of Account class)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Address
{
    String alias;
    Account account;

    .. (remainder of Address class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public String alias; // Same name as real field above
        public Account.PK account; // Same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s,"::");
            this.alias = Long.valueOf(token.nextToken()).longValue();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            this.account = new Account.PK(token.nextToken());
        }

        public String toString()
        {
            return alias + "::" + this.account.toString();
        }

        public int hashCode()
        {
            return alias.hashCode() ^ account.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.alias.equals(this.alias) &&
this.account.equals(otherPK.account);
            }
            return false;
        }
    }
}
```

To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
    <entity class="mydomain.Account">
        <table name="ACCOUNT"/>
        <id-class class="mydomain.Account.PK"/>
        <attributes>
            <id name="id">
                <column name="ACCOUNT_ID"/>
            </id>
            <basic name="firstName">
                <column name="FIRSTNAME" length="50"/>
            </basic>
            <basic name="secondName">
                <column name="LASTNAME" length="50"/>
            </basic>
            <one-to-many name="addresses" mapped-by="account">
                <map-key name="alias"/>
            </one-to-many>
    </entity>

    <entity class="mydomain.Address">
        <table name="ADDRESS"/>
        <id-class class="mydomain.Address.PK"/>
        <attributes>
            <id name="account">
                <column name="ACCOUNT_ID"/>
            </id>
            <id name="alias">
                <column name="KEY"/>
            </id>
            <basic name="city">
```

```
                <column name="CITY"/>
            </basic>
            <basic name="street">
                <column name="STREET"/>
            </basic>
            <many-to-one name="account"/>
        </attributes>
    </entity>
</entity-mappings>
```

So now we have the following datastore schema

Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- If we had omitted the "alias" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "alias" field too as part of the PK.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.1 JDO API

JDO API

The work in this section can be split into several sections.

- With JDO you need to create a PersistenceManagerFactory to connect to a datastore
- With JDO you need to create a PersistenceManager to provide the interface to persisting/accessing objects
- Controlling the transaction
- Accessing persisted object via queries, using JDOQL, or SQL

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.2 Persistence Manager Factory

Persistence Manager Factory

Any JDO-enabled application will require at least one PersistenceManagerFactory. Typically applications create one per datastore being utilised. A PersistenceManagerFactory provides access to PersistenceManagers which allow objects to be persisted, and retrieved. The PersistenceManagerFactory can be configured to provide particular behaviour.

The simplest way of creating a PersistenceManagerFactory is as follows

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("javax.jdo.option.ConnectionDriverName","com.mysql.jdbc.Driver");
properties.setProperty("javax.jdo.option.ConnectionURL","jdbc:mysql://localhost/myDB");
properties.setProperty("javax.jdo.option.ConnectionUserName","login");
properties.setProperty("javax.jdo.option.ConnectionPassword","password");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

A slight variation on this, is to use a file to specify these properties like this

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/myDB
javax.jdo.option.ConnectionUserName=login
javax.jdo.option.ConnectionPassword=password
```

and then to create the PersistenceManagerFactory using this file

```
File propsFile = new File(filename);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(propsFile);
```

Another alternative would be to call JDOHelper.getPersistenceManagerFactory(jndiLocation, context);, hence accessing the properties via JNDI.

Whichever way we wish to obtain the PersistenceManagerFactory we have defined a series of properties to give the behaviour of the PersistenceManagerFactory. The first property specifies to use the DataNucleus implementation, and the following 4 properties define the datastore that it should connect to. There are many properties available. Some of these are standard JDO properties, and some are DataNucleus extensions.

JDO Persistence Properties

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Name	Values	Description
javax.jdo.PersistenceManagerFactoryClass		The name of the PersistenceManager implementation. *org.datanucleus.jdo.JDOPersistenceManagerFactory*
javax.jdo.option.ConnectionFactory		Alias for datanucleus.ConnectionFactory
javax.jdo.option.ConnectionFactory2		Alias for datanucleus.ConnectionFactory2
javax.jdo.option.ConnectionFactoryName		Alias for datanucleus.ConnectionFactoryName
javax.jdo.option.ConnectionFactory2Name		Alias for datanucleus.ConnectionFactory2Name
javax.jdo.option.ConnectionDriverName		Alias for datanucleus.ConnectionDriverName
javax.jdo.option.ConnectionURL		Alias for datanucleus.ConnectionURL
javax.jdo.option.ConnectionUserName		Alias for datanucleus.ConnectionUserName
javax.jdo.option.ConnectionPassword		Alias for datanucleus.ConnectionPassword
javax.jdo.option.IgnoreCache	true | false	Alias for datanucleus.IgnoreCache
javax.jdo.option.Multithreaded	true | false	Alias for datanucleus.Multithreaded
javax.jdo.option.NontransactionalRead	true | false	Alias for datanucleus.NontransactionalRead
javax.jdo.option.NontransactionalWrite	true | false	Alias for datanucleus.NontransactionalWrite
javax.jdo.option.Optimistic	true | false	Alias for datanucleus.Optimistic
javax.jdo.option.RetainValues	true | false	Alias for datanucleus.RetainValues
javax.jdo.option.RestoreValues	true | false	Alias for datanucleus.RestoreValues
javax.jdo.option.Mapping		Alias for datanucleus.Mapping *RDBMS datastores only*
javax.jdo.mapping.Catalog		Alias for datanucleus.Catalog *RDBMS datastores only*
javax.jdo.mapping.Schema		Alias for datanucleus.Schema *RDBMS datastores only*
javax.jdo.option.DetachAllOnCommit	true | false	Alias for datanucleus.DetachAllOnCommit
javax.jdo.option.TransactionType		Alias for datanucleus.TransactionType This is new in JDO2.1
javax.jdo.option.PersistenceUnitName		Alias for datanucleus.PersistenceUnitName This is new in JDO2.1
javax.jdo.option.ServerTimeZoneID		Alias for datanucleus.ServerTimeZoneID This is new in JDO2.1
javax.jdo.option.Name		Name of the PMF to use. This is for use with "named PMF" functionality in JDO 2.1 referring to a PMF defined in "jdoconfig.xml".
javax.jdo.option.CopyOnAttach	true | false	Alias for datanucleus.CopyOnAttach This is new in JDO2.1
javax.jdo.option.ReadOnly	true | false	Alias for datanucleus.readOnlyDatastore This is new in JDO2.2

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Name	Values	Description
javax.jdo.option.TransactionIsolationLevel		Alias for datanucleus.transactionIsolation This is new in JDO2.2
javax.jdo.option.QueryTimeoutMillis		Alias for datanucleus.query.timeout This is new in JDO2.3

DataNucleus provides many properties to extend the control that JDO gives you. These can be used alongside teh above standard JDO properties, but will only work with DataNucleus. Please consult the Persistence Properties Guide for full details.

Named PMFs

Typically applications create one PMF per datastore being utilised. JDO2.1+ provides a way of creating a PMF with a particular name and set of properties. This utilises a configuration file that defines the named PMFs. This configuration file is called *jdoconfig.xml*, and is located under "META-INF/". Let's see an example of a *jdoconfig.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<jdoconfig xmlns="http://java.sun.com/xml/ns/jdo/jdoconfig"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://java.sun.com/xml/ns/jdo/jdoconfig">

    <!-- Datastore Txn PMF -->
    <persistence-manager-factory name="Datastore">
        <property name="javax.jdo.PersistenceManagerFactoryClass"
            value="org.datanucleus.jdo.JDOPersistenceManagerFactory"/>
        <property name="javax.jdo.option.ConnectionDriverName"
            value="com.mysql.jdbc.Driver"/>
        <property name="javax.jdo.option.ConnectionURL"
            value="jdbc:mysql://localhost/datanucleus?useServerPrepStmts=false"/>
        <property name="javax.jdo.option.ConnectionUserName"
            value="datanucleus"/>
        <property name="javax.jdo.option.ConnectionPassword"
            value=""/>
        <property name="javax.jdo.option.Optimistic"
            value="false"/>
        <property name="datanucleus.autoCreateSchema"
            value="true"/>
    </persistence-manager-factory>

    <!-- Optimistic Txn PMF -->
    <persistence-manager-factory name="Optimistic">
        <property name="javax.jdo.PersistenceManagerFactoryClass"
            value="org.datanucleus.jdo.JDOPersistenceManagerFactory"/>
        <property name="javax.jdo.option.ConnectionDriverName"
            value="com.mysql.jdbc.Driver"/>
        <property name="javax.jdo.option.ConnectionURL"
            value="jdbc:mysql://localhost/datanucleus?useServerPrepStmts=false"/>
        <property name="javax.jdo.option.ConnectionUserName"
            value="datanucleus"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <property name="javax.jdo.option.ConnectionPassword"
            value=""/>
        <property name="javax.jdo.option.Optimistic"
            value="true"/>
        <property name="datanucleus.autoCreateSchema"
            value="true"/>
    </persistence-manager-factory>

</jdoconfig>
```

So in this example we have 2 named PMFs. The first is known by the name "Datastore" and utilises datastore transactions. The second is known by the name "Optimistic" and utilises optimistic transactions. You simply define all properties for the particular PMF within its specification block. And finally we instantiate our PMF like this

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("Optimistic");
```

That's it. The PMF we are returned from JDOHelper will have all of the properties defined in *jdoconfig.xml* under the name of "Optimistic".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.3 Persistence Manager

Persistence Manager

As you read in the guide for PersistenceManagerFactory, to control the persistence of your objects you will require at least one PersistenceManagerFactory. Once you have obtained this object you then use this to obtain a PersistenceManager. A PersistenceManager provides access to the operations for persistence of your objects. This short guide will demonstrate some of the more common operations.

You obtain a PersistenceManager as follows

```
PersistenceManager pm = pmf.getPersistenceManager();
```

In general you will be performing all operations on a PersistenceManager within a transaction, whether your transactions are controlled by your J2EE container, by a framework such as Spring, or by locally defined transactions. In the examples below we will omit the transaction demarcation for clarity.

Persisting an Object

The main thing that you will want to do with the data layer of a JDO-enabled application is persist your objects into the datastore. As we mentioned earlier, a PersistenceManagerFactory represents the datastore where the objects will be persisted. So you create a normal Java object in your application, and you then persist this as follows

```
pm.makePersistent(obj);
```

This will result in the object being persisted into the datastore, though clearly it will not be persistent until you commit the transaction. The LifecycleState of the object changes from Transient to PersistentClean (after makePersistent), to Hollow (at commit).

Using an Objects identity

Once you have persisted an object, it has an "identity". This is a unique way of identifying it. You can obtain the identity by calling

```
Object id = pm.getObjectId(obj);
```

So what ? Well the identity can be used to retrieve the object again at some other part in your application. So you pass the identity into your application, and the user clicks on some button on a web page and that button corresponds to a particular object identity. You can then go back to your data layer and retrieve the object as follows

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Object obj = pm.getObjectById(id);
```

A DataNucleus extension is to pass in a String form of the identity to the above method. It accepts identity strings of the form

- *{fully-qualified-class-name}:{key}*
- *{discriminator-name}:{key}*

where the *key* is the identity value (datastore-identity) or the result of PK.toString() (application-identity). So for example we could input

```
obj = pm.getObjectById("mydomain.MyClass:3");
```

Deleting an Object

When you need to delete an object that you had previous persisted, deleting it is simple. Firstly you need to get the object itself, and then delete it as follows

```
Object obj = pm.getObjectById(id);  // Retrieves the object to delete
pm.deletePersistent(obj);
```

Modifying a persisted Object

To modify a previously persisted object you only need to enlist the (updated) object in a transaction and its changes will be propagated to the datastore at commit of the transaction.

Detaching a persisted Object

You often have a previously persisted object and you want to use it away from the data-access layer of your application. In this case you want to detach the object (and its related objects) so that they can be passed across to the part of the application that requires it. To do this you do

```
Object detachedObj = pm.detachCopy(obj); // Returns a copy of the persisted object,
in detached state
```

The detached object is like the original object except that it has no StateManager connected, and it stores its JDO identity and version. It retains a list of all fields that are modified while it is detached. This means that when you want to "attach" it to the data-access layer it knows what to update.

As an alternative, to make the detachment process transparent, you can set the PMF property *datanucleus.DetachAllOnCommit* to true and when you commit your transaction all objects enlisted in the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

transaction will be detached.

Attaching a persisted Object

You've detached an object (shown above), and have modified it in your application, and you now want to attach it back to the persistence layer. You do this as follows

```
Object attachedObj = pm.makePersistent(obj); // Returns a copy of the detached
object, in attached state
```

Refresh of objects

In the situation where you have an object and you think that its values may have changed in the datastore you can update its values to the latest using the following

```
pm.refresh(obj);
```

What this will do is as follows

- Refresh the values of all FetchPlan fields in the object
- Unload all non-FetchPlan fields in the object

If the object had any changes they will be thrown away by this step, and replaced by the latest datastore values.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.4 Persistence Manager Proxy

Persistence Manager Proxies

As you read in the guide for PersistenceManager, you perform all operations using a *PersistenceManager*. This means that you need to obtain this when you want to start datastore operations. In some architectures (e.g in a web environment) it can be convenient to maintain a single PersistenceManager for use in a servlet init() method to initialise a static variable. Alternatively for use in a SessionBean to initialise a static variable. Thereafter you just refer to the proxy. The proxy isn't the actual *PersistenceManager* just a proxy, delegating to the real object. If you call close() on the proxy the real PM will be closed, and when you next invoke an operation on the proxy it will create a new PM delegate and work with that.

To create a PM proxy is simple

```
PersistenceManager pm = pmf.getPersistenceManagerProxy();
```

So we have our proxy, and now we can perform operations

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.5 Auto-Start

Automatic Startup

By default with JDO implementations when you open a PersistenceManagerFactory and obtain a PersistenceManager DataNucleus knows nothing about which classes are to be persisted to that datastore. JDO implementations only load the Meta-Data for any class when the class is first enlisted in a PersistenceManager operation. For example you call makePersistent on an object. The first time a particular class is encountered DataNucleus will dynamically load the Meta-Data for that class. This typically works well since in an application in a particular operation the PersistenceManagerFactory may well not encounter all classes that are persistable to that datastore. The reason for this dynamic loading is that JDO implementations can't be expected to scan through the whole Java CLASSPATH for classes that could be persisted there. That would be inefficient.

There are situations however where it is desirable for DataNucleus to have knowledge about what is to be persisted, so that it can load the Meta-Data at initialisation of the persistence factory and hence when the classes are encountered for the first time nothing needs doing. There are two ways of providing this

- Define your classes/MetaData in a Persistence Unit and when the PersistenceManagerFactory is initialised it loads the persistence unit, and hence the MetaData for the defined classes and mapping files. This is described on the linked page
- Use a DataNucleus extension known as Auto-Start Mechanism. This is set with the persistence property datanucleus.autoStartMechanism. This can be set to None, XML, Classes, MetaData. These are described below. Please note that other plugins, such as RDBMS provide alternative options.

None

With this property set to "None" DataNucleus will have no knowledge about classes that are to be persisted into that datastore and so will add the classes when the user utilises them in calls to the various PersistenceManager methods.

XML

With XML, DataNucleus stores the information for starting up DataNucleus in an XML file. This is, by default, located in datanucleusAutoStart.xml in the current working directory. The file name can be configured using the persistence factory property datanucleus.autoStartMechanismXmlFile. The file is read at startup and DataNucleus loads the classes using this information.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from the XML configuration file and finds that a table/class required by the this file data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property datanucleus.autoStartMechanismMode defines the behaviour of DataNucleus for this situation.

- Checked will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- Quiet (the default) will simply remove the entry from the XML file and continue without exception.
- Ignored will simply continue without doing anything.

Classes

With Classes, the user provides to the persistence factory the list of classes to use as the initial list of classes to be persisted. They specify this via the persistence property datanucleus.autoStartClassNames, specifying the list of classes as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

MetaData

With MetaData, the user provides to the persistence factory the list of metadata files to use as the initial list of classes to be persisted. They specify this via the persistence property datanucleus.autoStartMetaDataFiles, specifying the list of metadata files as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.6 Datastore Control

Datastore Control

Sometimes a datastore has particular requirements, and maybe is out of direct control of an application being managed by a different group. In this situation you may have to configure the datastore layer to meet these requirements.

Read-Only

If your datastore is read-only (you can't add/update/delete any data in it), obviously you could just configure your application to not perform these operations. An alternative is to set the PersistenceManagerFactory as "read-only". You do this by setting the persistence property **javax.jdo.option.ReadOnly** to *true*.

From now on, whenever you perform a persistence operation that implies a change in datastore data, the JDO operation will throw a *JDOReadOnlyException*. **This is included from JDO 2.2 onwards** (but has been in DataNucleus and JPOX before it for a long time).

DataNucleus provides an additional control over the behaviour when an attempt is made to change a read-only datastore. The default behaviour is to throw an exception. You can change this using the persistence property *datanucleus.readOnlyDatastoreAction* with values of "EXCEPTION" (default), and "IGNORE". "IGNORE" has the effect of simply ignoring all attempted updates to readonly objects.

Fixed Schema

Some datastores have a "schema" defining the structure of data stored within that datastore. In this case we don't want to allow updates to the structure at all. You can set this when creating your PersistenceManagerFactory by setting the persistence property **datanucleus.fixedDatastore** to *true*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.7 Datastore Replication

Datastore Replication

Many applications make use of multiple datastores. It is a common requirement to be able to replicate parts of one datastore in another datastore. Obviously, depending on the datastore, you could make use of the datastores own capabilities for replication. DataNucleus provides its own extension to JDO to allow replication from one datastore to another. This extension doesn't restrict you to using 2 datastores of the same type. You could replicate from RDBMS to XML for example, or from MySQL to HSQLDB.

The following sample code will replicate all objects of type *Product* and *Employee* from PMF1 to PMF2. These PMFs are created in the normal way so, as mentioned above, PMF1 could be for a MySQL datastore, and PMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```
import org.datanucleus.jdo.JDOReplicationManager;

...

JDOReplicationManager replicator = new JDOReplicationManager(pmf1, pmf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.8 Transaction Types

Transaction Types

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. There are the following types of transaction.

- Transactions can lock all records in a datastore and keep them locked until they are ready to commit their changes. These are known as Pessimistic (or datastore) Transactions.
- Transactions can simply assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. These are known as Optimistic Transactions.
- Non-transactional where you perform operations effectively in "auto-commit" mode

Pessimistic (Datastore) Transactions

Pessimistic transactions are the default in JDO. They are suitable for short lived operations where no user interaction is taking place and so it is possible to block access to datastore entities for the duration of the transaction.

By default DataNucleus does not currently lock the objects fetched in a pessimistic transaction, but you can configure this behaviour for RDBMS datastores by setting the persistence property datanucleus.SerializeRead to true. This will result in all "SELECT ... FROM ..." statements being changed to be "SELECT ... FROM ... FOR UPDATE". This will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax. This can be done on a transaction-by-transaction basis by doing

```
org.datanucleus.jdo.JDOTransaction tx =
    (org.datanucleus.jdo.JDOTransaction)pm.currentTransaction();
tx.setSerializeRead(true);
```

Alternatively, on a per query basis, you would do

```
org.datanucleus.jdo.JDOQuery jdoQuery =
    (org.datanucleus.jdo.JDOQuery)pm.newQuery(...);
jdoQuery.setSerializeRead(true);
```

With a pessimistic transaction DataNucleus will grab a datastore connection at the first operation, and maintain it for the duration of the transaction. A single connection is used for the transaction (with the exception of any Identity Generation operations which need datastore access, so these can use their own connection).

In terms of the process of a pessimistic (datastore) transaction, we demonstrate this below.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Operation	DataNucleus process	Datastore process
Start transaction		
Persist object	Prepare object (1) for persistence	Open connection. Insert the object (1) into the datastore
Update object	Prepare object (2) for update	Update the object (2) into the datastore
Persist object	Prepare object (3) for persistence	Insert the object (3) into the datastore
Update object	Prepare object (4) for update	Update the object (4) into the datastore
Flush	No outstanding changes so do nothing	
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	Insert the object (5) into the datastore
Update object	Prepare object (6) for update	Update the object (6) into the datastore
Commit transaction		Commit connection

So here whenever an operation is performed, DataNucleus pushes it straight to the datastore. Consequently any queries will always reflect the current state of all objects in use. However this mode of operation has no version checking of objects and so if they were updated by external processes in the meantime then they will overwrite those changes.

It should be noted that DataNucleus provides two persistence properties that allow an amount of control over when flushing happens with datastore transactions.

- *datanucleus.datastoreTransactionDelayOperations* when set to true will try to delay all datastore operations until commit/flush.
- *datanucleus.datastoreTransactionFlushLimit* represents the number of dirty objects before a flush is performed. This defaults to 1.

Optimistic Transactions

Optimistic transactions are the other option in JDO. They are suitable for longer lived operations maybe where user interaction is taking place and where it would be undesirable to block access to datastore entities for the duration of the transaction. The assumption is that data altered in this transaction will not be updated by other transactions during the duration of this transaction, so the changes are not propagated to the datastore until commit()/flush(). The data is checked just before commit to ensure the integrity in this respect. The most convenient way of checking data for updates is to maintain a column on each table that handles optimistic transaction data. The user will decide this when generating their MetaData.

Rather than placing version/timestamp columns on all user datastore tables, JDO2 allows the user to notate particular classes as requiring optimistic treatment. This is performed by specifying in MetaData or annotations the details of the field/column to use for storing the version - see versioning for JDO. With JDO the version is added in a surrogate column, whereas a vendor extension allows you to have a field in

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

your class ready to store the version.

In JDO2 the version is stored in a surrogate column in the datastore so it also provides a method for accessing the version of an object. You can call *JDOHelper.getVersion(object)* and this returns the version as an Object (typically Long or Timestamp). This will return null for a transient object, and will return the version for a persistent object. If the object is not *PersistenceCapable* then it will also return null.

In terms of the process of an optimistic transaction, we demonstrate this below.

Operation	DataNucleus process	Datastore process
Start transaction		
Persist object	Prepare object (1) for persistence	
Update object	Prepare object (2) for update	
Persist object	Prepare object (3) for persistence	
Update object	Prepare object (4) for update	
Flush	Flush all outstanding changes to the datastore	Open connection. Version check of object (1) Insert the object (1) in the datastore. Version check of object (2) Update the object (2) in the datastore. Version check of object (3) Insert the object (3) in the datastore. Version check of object (4) Update the object (4) in the datastore.
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	
Update object	Prepare object (6) for update	
Commit transaction	Flush all outstanding changes to the datastore	Version check of object (5) Insert the object (5) in the datastore Version check of object (6) Update the object (6) in the datastore. Commit connection.

Here no changes make it to the datastore until the user either commits the transaction, or they invoke flush(). The impact of this is that when performing a query, by default, the results may not contain the modified objects unless they are flushed to the datastore before invoking the query. Depending on whether you need the modified objects to be reflected in the results of the query governs what you do about that. If you invoke flush() just before running the query the query results will include the changes. The obvious benefit of optimistic transaction is that all changes are made in a block and version checking of objects is performed before application of changes, hence this mode copes better with external processes updating the objects.

See also :-

- JDO MetaData reference for <version> element
- JDO Annotations reference for @Version

No Transactions

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DataNucleus allows the ability to operate without transactions. With JDO this can be enabled by setting the 2 properties datanucleus.NontransactionalRead, datanucleus.NontransactionalWrite to true. DataNucleus supports these allowing the ability to read objects and make updates outside of transactions. The important thing is that to use this mode of operation, you must enable it by setting the property (DataNucleus defaults to false if not set). Please be aware that any non transactional updates will typically be committed to the datastore **by the subsequent transaction**, with the updates being queued until that point.

Persistence-by-Reachability at commit()

When a transaction is committed JDO will, by default, run its reachability algorithm to check if any reachable objects have been persisted and are no longer reachable. If an object is found to be no longer reachable and was only persisted by being reachable (not by an explicit persist operation) then it will be removed from the datastore. You can turn off this reachability check for JDO by setting the persistence property **datanucleus.persistenceByReachabilityAtCommit** to false.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.9 Transactions

JDO Transactions

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. Transactions can be managed by the users application, or can be managed by a framework (such as Spring), or can be managed by a J2EE container. These are described below.

- Local transactions : managed using the JDO Transaction API
- JTA transactions : managed using the JTA UserTransaction API, or using the JDO Transaction API
- Container-managed transactions : managed by a J2EE environment

See also :-

- Transaction Types
- JDO Datastore Connections

Locally Managed Transactions

When using a JDO implementation such as DataNucleus in a J2SE environment, the transactions are by default Locally Managed Transactions. The users code will manage the transactions by starting, and commiting the transaction itself. With these transactions with JDO Javadoc you would do something like

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    {users code to persist objects}

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
pm.close();
```

When you use a framework like Spring you would not need to specify the tx.begin(), tx.commit(), tx.rollback() since that would be done for you. The basic idea with Locally Managed transactions is that you are managing the transaction start and end.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

JTA Transactions

When using a JDO implementation such as DataNucleus in a J2SE environment, you can also make use of JTA Transactions. **You define the persistence property *javax.jdo.option.TransactionType* setting it to "JTA"**. Then you make use of JTA (or JDO) to demarcate the transactions. So you could do something like

```
UserTransaction ut = (UserTransaction)
    new InitialContext().lookup("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
try
{
    ut.begin();

    {users code to persist objects}

    ut.commit();
}
finally
{
    pm.close();
}
```

So here we used the JTA API to begin/commit the controlling (*javax.transaction.UserTransaction*).

An alternative is where you don't have a UserTransaction started and just use the JDO API, which will start the UserTransaction for you.

```
UserTransaction ut = (UserTransaction)
    new InitialContext().lookup("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin(); // Starts the UserTransaction

    {users code to persist objects}

    tx.commit(); // Commits the UserTransaction
}
finally
{
    pm.close();
}
```

Important : please note that you need to set both transactional and nontransactional datasources, and **the nontransactional cannot be JTA**.

Container Managed Transactions

When using a J2EE container you are giving over control of the transactions to the container. Here you

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

have Container Managed Transactions. In terms of your code, you would do like the previous example except that you would OMIT the tx.begin(), tx.commit(), tx.rollback() since the J2EE container will be doing this for you.

Transaction Isolation

JDO 2.2 provides a mechanism for specification of the transaction isolation level. This can be specified globally via the PersistenceManagerFactory property datanucleus.transactionIsolation (javax.jdo.option.TransactionIsolationLevel). It accepts the following values

- **read-uncommitted** : dirty reads, non-repeatable reads and phantom reads can occur
- **read-committed** : dirty reads are prevented; non-repeatable reads and phantom reads can occur
- **repeatable-read** : dirty reads and non-repeatable reads are prevented; phantom reads can occur
- **serializable** : dirty reads, non-repeatable reads and phantom reads are prevented

The default (in DataNucleus) is read-committed. An attempt to set the isolation level to an unsupported value (for the datastore) will throw a JDOUserException. For example DB4O only supports *read-committed* so will always use that. As an alternative you can also specify it on a per-transaction basis as follows (using the names above).

```
Transaction tx = pm.currentTransaction();
...
tx.setIsolationLevel("read-committed");
```

JDO Transaction Synchronisation

There are situations where you may want to get notified that a transaction is in course of being committed or rolling back. To make that happen, you would do something like

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    tx.setSynchronization(new javax.transaction.Synchronization()
    {
        public void beforeCompletion()
        {
             // before commit or rollback
        }

        public void afterCompletion(int status)
        {
            if (status == javax.transaction.Status.STATUS_ROLLEDBACK)
            {
                // rollback
            }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            else if (status == javax.transaction.Status.STATUS_COMMITTED)
            {
                // commit
            }
        }
    });

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
pm.close();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.10 Cache

JDO Caching

Caching is an essential mechanism in providing efficient usage of resources in many systems. Data management using JDO is no different and provides a definition of caching at 2 levels. Caching allows objects to be retained and returned rapidly without having to make an extra call to the datastore. The 2 levels of caching available with DataNucleus are

- Level 1 Cache - mandated by the JDO specification, and represents the caching of instances within a PersistenceManager
- Level 2 Cache - represents the caching of instances within a PersistenceManagerFactory (across multiple PersistenceManager's)

You can think of a cache as a Map, with values referred to by keys. In the case of JDO, the key is the object identity (identity is unique in JDO).

Level 1 Cache

The Level 1 Cache is always enabled. There are inbuilt types for the Level 1 Cache available for selection.

DataNucleus supports the following types of Level 1 Cache.

- weak - uses a weak reference backing map. If JVM garbage collection clears the reference, then the object is removed from the cache. This is the default Level 1 Cache
- soft - uses a soft reference backing map. If the map entry value object is not being actively used, then garbage collection may garbage collect the reference, in which case the object is removed from the cache.
- hard - uses a normal HashMap backing. With this option all references are strong meaning that objects stay in the cache until they are explicitly removed by calling remove() on the cache.

You can specify the type of Level 1 Cache by providing the PMF property datanucleus.cache.level1.type. You set this to the value of the type required. If you want to remove objects from the L1 cache programmatically you should use pm.evict() or pm.evictAll().

Objects are placed in the L1 cache (and updated there) during the course of the transaction. This provides rapid access to the objects in use in the users application and is used to guarantee that there is only one object with a particular identity at any one time for that PersistenceManager.

Plugin The Level 1 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it.

Level 2 Cache

By default in DataNucleus, Level 2 Cache is disabled. The user can configure the Level 2 Cache if they so wish. This is controlled by use of the persistence property datanucleus.cache.level2 which is a boolean property. There are builtin Level 2 caches available for use. You can specify the type of Level 2 Cache by providing the persistence property datanucleus.cache.level2.type. You set this to type of cache required.

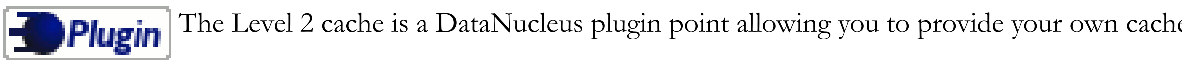
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

With the Level 2 Cache you currently have the following options.

- default - the default option. Provides support for the JDO 2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Weak references are held to non pinned objects.
- soft - Provides support for the JDO 2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Soft references are held to non pinned objects.
- EHCache - a simple wrapper to EHCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- EHCacheClassBased - similar to the EHCache option but class-based.
- OSCache - a simple wrapper to OSCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- SwarmCache - a simple wrapper to SwarmCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- Oracle Coherence - a simple wrapper to Oracle's Coherence caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning. Oracle's caches support distributed caching, so you could, in principle, use DataNucleus in a distributed environment with this option.

The EHCache, OSCache, SwarmCache and Coherence caches are available in the datanucleus-cache plugin.

From Access Platform 1.0 M3 onwards objects are placed in the L2 cache when you commit() the transaction of a PersistenceManager. This means that you only have datastore-persisted objects in that cache. Also, if an object is deleted during a transaction then at commit it will be removed from the L2 cache if it is present.



The Level 2 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it. Use the examples of the EHCache, Coherence caches etc as reference.

Controlling the Level 2 Cache

The majority of times when using a JDO-enabled system you will not have to take control over any aspect of the caching other than specification of whether to use a Level 2 Cache or not. With JDO2 and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the PersistenceManagerFactory.

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
```

The DataStoreCache interface provides methods to control the retention of objects in the cache. You have 3 groups of methods

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- evict - used to remove objects from the Level 2 Cache
- pin - used to pin objects into the cache, meaning that they will not get removed by garbage collection, and will remain in the Level 2 cache until removed.
- unpin - used to reverse the effects of pinning an object in the Level 2 cache. This will mean that the object can thereafter be garbage collected if not being used.

These methods can be called to pin objects into the cache that will be much used. Clearly this will be very much application dependent, but it provides a mechanism for users to exploit the caching features of JDO. If an object is not "pinned" into the L2 cache then it can typically be garbage collected at any time, so you should utilise the pinning capability for objects that you wish to retain access to during your application lifetime. For example, if you have an object that you want to be found from the cache you can do

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
cache.pinAll(MyClass.class, false); // Pin all objects of type MyClass from now on
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    pm.makePersistent(myObject);
    // "myObject" will now be pinned since we are pinning all objects of type
MyClass.

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
            tx.close();
    }
}
```

Thereafter, whenever something refers to myObject, it will find it in the Level 2 cache. To turn this behaviour off, the user can either unpin it or evict it.

JDO2.2

JDO2.2 allows control over which classes are put into a Level 2 cache. You do this by specifying the **cacheable** attribute to *false* (defaults to true). [For previous versions of JDO you can use the DataNucleus extension "cacheable" on the class]. So with the following specification, no objects of type *MyClass* will be put in the L2 cache.

```
Using XML:
<class name="MyClass" cacheable="false">
    ...
</class>

Using Annotations:
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
@Cacheable("false")
public class MyClass
{
    ...
}
```

JDO2.2

JDO2.2 allows you control over which fields of an object are put in the Level 2 cache. You do this by specifying the **cacheable** attribute to *false* (defaults to true). [For previous versions of JDO you can use the DataNucleus extension "cacheable" on the field]. This setting is only required for fields that are relationships to other persistable objects. Like this

```
Using XML:
<class name="MyClass">
    <field name="values"/>
    <field name="elements" cacheable="false"/>
    ...
</class>

Using Annotations:
public class MyClass
{
    ...

    Collection values;

    @Cacheable("false")
    Collection elements;
}
```

So in this example we will cache "values" but not "elements". If a field is *cacheable* then

- If it is a persistable object, the "identity" of the related object will be stored in the Level 2 cache for this field of this object
- If it is a Collection of persistable elements, the "identity" of the elements will be stored in the Level 2 cache for this field of this object
- If it is a Map of persistable keys/values, the "identity" of the keys/values will be stored in the Level 2 cache for this field of this object
- If a persistable related object is embedded or serialised then it **will not be cached**. It is hoped to support this in a later release.

When pulling an object in from the Level 2 cache and it has a reference to another object Access Platform uses the "identity" to find that object in the Level 1 or Level 2 caches to re-relate the objects.

L2 Cache using Oracle Coherence

DataNucleus provides a simple wrapper to Oracle's Coherence caches. This currently takes the NamedCache interface in Coherence and instantiates a cache of a user provided name. To enabled this you should set the following persistence properties

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.cache.level2=true
datanucleus.cache.level2.type=coherence
datanucleus.cache.level2.cacheName={coherence cache name}
```

The Coherence cache name is the name that you would normally put into a call to CacheFactory.getCache(name). As mentioned earlier, this cache does not support the pin/unpin operations found in the standard JDO 2 interface. However you do have the benefits of Oracle's distributed/serialized caching. If you require more control over the Coherence cache whilst using it with DataNucleus, you can just access the cache directly via

```
DataStoreCache cache = pmf.getDataStoreCache();
NamedCache tangosolCache = ((TangosolLevel2Cache)cache).getTangosolCache();
```

L2 Cache using EHCache

DataNucleus provides a simple wrapper to EHCache's caches. To enable this you should set the persistence properties

```
    datanucleus.cache.level2=true
    datanucleus.cache.level2.type=ehcache
    datanucleus.cache.level2.cacheName={cache name}
    datanucleus.cache.level2.configurationFile={EHCache configuration file (in
classpath)}
```

The EHCache plugin also provides an alternative L2 Cache that is class-based. To use this you would need to replace "ehcache" above with "ehcacheclassbased".

L2 Cache using OSCache

DataNucleus provides a simple wrapper to OSCache's caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2=true
datanucleus.cache.level2.type=oscache
datanucleus.cache.level2.cacheName={cache name}
```

L2 Cache using SwarmCache

DataNucleus provides a simple wrapper to SwarmCache's caches. To enable this you should set the persistence properties

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.cache.level2=true
datanucleus.cache.level2.type=swarmcache
datanucleus.cache.level2.cacheName={cache name}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.11 Datastore Connection

Datastore Connections

DataNucleus utilises connections to the datastore differently depending on the transaction type.

For pessimistic (or datastore) transactions when begin() is called on the transaction, a connection will be obtained to the datastore. This datastore connection will be held for the duration of the transaction until such time as either commit() or rollback() are called.

For optimistic transactions things are a little more complicated. The call to begin() has no effect with respect to datastore connections. All updates to the datastore are delayed until flush() or commit() and so no connection is obtained until that time. When flush() is called, or the transaction committed a datastore connection is finally obtained and it is held open until commit/rollback completes. when a datastore operation is required. The connection is typically released after performing that operation. So datastore connections, in general, are held for much smaller periods of time. This is complicated slightly by use of the PMF property java.jdo.IgnoreCache. When this is set to false, the connection, once obtained, is not released until the call to commit()/rollback().

When the operation is outside of a transaction, where nontransactionalRead is enabled, the Connection is allocated when the operation starts, and closed when it ends for the particular PersistenceManager. This can result in "ConnectionInUse" problems where another operation on another thread comes in and tries to perform something while that first operation is still in use. This happens because the JDO spec requires an implementation to use a single datastore connection at any one time. When this situation crops up the user ought to use multiple PersistenceManagers.

Another important aspect is use of queries for Optimistic transactions, or for non-transactional contexts. In these situations it isn't possible to keep the datastore connection open indefinitely and so when the Query is executed the ResultSet is then read into core making the queried objects available thereafter.

Occasionally systems may need access to the datastore directly. JDO2 provides an accessor for obtaining a connection to the datastore. You obtain a connection as follows

User Connection

JDO2 defines a mechanism for users to access the native connection to the datastore, so that they can perform other operations as necessary. You obtain a connection as follows (for RDBMS)

```
// Obtain the connection from the JDO implementation
JDOConnection conn = pm.getDataStoreConnection();
try
{
    java.sql.Connection sqlConn = (java.sql.Connection)conn;

    ... use the "sqlConn" connection to perform some operations.
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
finally
{
    // Hand the connection back to the JDO implementation
    conn.close();
}
```

and for DB4O this would be

```
// Obtain the connection from the JDO implementation
JDOConnection conn = pm.getDataStoreConnection();
try
{
    com.db4o.ObjectContainer objContainer =
(com.db4o.ObjectContainer)conn.getNativeConnection();

    ... use the "objContainer" connection to perform some operations.
}
finally
{
    // Hand the connection back to the JDO implementation
    conn.close();
}
```

The "JDOConnection" Javadoc in the case of DataNucleus is a wrapper to the native connection for the type of datastore being used. You now have a connection allowing direct access to the datastore. Things to bear in mind with this connection

- You must return the connection back to the PersistenceManager before performing any JDO PM operation. You do this by calling conn.close()
- If you don't return the connection and try to perform a JDO PM operation which requires the connection then a JDOUserException is thrown.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.12 Attach/Detach

JDO Attach/Detach

JDO provides an interface to the persistence of objects. JDO 1.0 doesn't provide a way of taking an object that was just persisted and just work on it and update the persisted object later. The user has to copy the fields manually and copy them back to the persisted object later. JDO 2.0 introduces a new way of handling this situation, by detaching an object from the persistence graph, allowing it to be worked on in the users application. It can then be attached to the persistence graph later. The first thing to do to use a class with this facility is to tag it as "detachable". This is done by adding the attribute

```
<class name="MyClass" detachable="true">
```

This acts as an instruction to the enhancement process to add methods necessary to utilise the attach/detach process.

The following code fragment highlights how to use the attach/detach mechanism

```
Product working_product=null;
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();

    Product prod=new Product(name,description,price);
    pm.makePersistent(prod);

    // Detach the product for use
    working_product = (Product)pm.detachCopy(prod);

    tx.commit();
}
catch (Exception e)
{
    // Handle the exception
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}

// Work on the detached object in our application
working_product.setPrice(new_price);

...

// Reattach the updated object
tx = pm.currentTransaction();
try
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    tx.begin();

    Product attached_product = pm.makePersistent(working_product);

    tx.commit();
}
catch (Exception e)
{
    // Handle the exception
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
```

So we now don't need to do any manual copying of object fields just using a simple call to detach the object, and then attach it again later. Here are a few things to note with attach/detach :-

- Calling *detachCopy* on an object that is not detachable will return a **transient** instance that is a COPY of the original, so use the COPY thereafter.
- Calling *detachCopy* on an object that is detachable will return a **detached** instance that is a COPY of the original, so use this COPY thereafter
- A detached object retain the id of its datastore entity. Detached objects should be used where you want to update the objects and attach them later (updating the associated object in the datastore. If you want to create copies of the objects in the datastore with their own identities you should use makeTransient instead of detachCopy.
- Calling detachCopy will detach all fields of that object that are in the current Fetch Group for that class for that PersistenceManager.
- By default the fields of the object that will be detached are those in the Default Fetch Group.
- You should choose your Fetch Group carefully, bearing in mind which object(s) you want to access whilst detached. Detaching a relation field will detach the related object as well.
- If you don't detach a field of an object, you cannot access the value for that field while the object is detached.
- If you don't detach a field of an object, you can update the value for that field while detached, and thereafter you can access the value for that field.
- Calling makePersistent will return an (attached) copy of the detached object. It will attach all fields that were originally detached, and will also attach any other fields that were modified whilst detached.

When attaching an object graph (using makePersistent()) DataNucleus will, by default, make a check if each detached object has been detached from this datastore (since they could have been detached from a different datastore). This clearly can cause significant numbers of additional datastore activity with a large object graph. Consequently we provide a PMF property *datanucleus.attachSameDatastore* which, when set to true, will omit these checks and assume that we are attaching to the same datastore they were detached

from.

To read more about attach/detach and how to use it with fetch-groups you can look at our Tutorial on DAO Layer design.

Detach All On Commit

JDO2 also provides a mechanism whereby all objects that were enlisted in a transaction are automatically detached when the transaction is committed. You can enable this in one of 3 ways. If you want to use this mode globally for all PersistenceManagers (PMs) from a PersistenceManagerFactory (PMF) you could either set the PMF property "datanucleus.DetachAllOnCommit", or you could create your PMF and call the PMF method setDetachAllOnCommit(true). If instead you wanted to use this mode only for a particular PM, or only for a particular transaction for a particular PM, then you can call the PM method setDetachAllOnCommit(true) before the commit of the transaction, and it will apply for all transaction commits thereafter, until turned off (setDetachAllOnCommit(false). Here's an example

```
// Create a PMF
...

// Create an object
MyObject my = new MyObject();

PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    // We want our object to be detached when it's been persisted
    pm.setDetachAllOnCommit(true);

    // Persist the object that we created earlier
    pm.makePersistent(my);

    tx.commit();
    // The object "my" is now in detached state and can be used further
}
finally
{
    if (tx.isActive)
    {
        tx.rollback();
    }
}
```

Copy On Attach

By default when you are attaching a detached object it will return an attached copy of the detached object. JDO2.1 provides a new feature that allows this attachment to just migrate the existing detached

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

object into attached state.

You enable this by setting the PersistenceManagerFactory (PMF) property datanucleus.CopyOnAttach to false. Alternatively you can use the methods *PersistenceManagerFactory.setCopyOnAttach(boolean flag)* or *PersistenceManager.setCopyOnAttach(boolean flag)*. If we return to the example at the start of this page, this now becomes

```
// Reattach the updated object
pm.setCopyOnAttach(false);
tx = pm.currentTransaction();
try
{
    tx.begin();

    // working product is currently in detached state

    pm.makePersistent(working_product);
    // working_product is now in persistent (attached) state

    tx.commit();
}
catch (Exception e)
{
    // Handle the exception
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
```

Please note that if you try to attach two detached objects representing the same underlying persistent object within the same transaction (i.e a persistent object with the same identity already exists in the level 1 cache), then a JDOUserException will be thrown.

Attach Policy

By default with JDO when you attach an object graph only the fields that have been changed since being detached will be updated in the datastore. Obviously, if performing replication of a datastore, this isn't of much use since you just want to detach from one datastore and attach into the other datastore. There is a persistence property **datanucleus.attachPolicy** (default=*attach-dirty*) that, if you set it to *attach-all* will attach all loaded/dirty fields.

Detach On Close

A backup to the above programmatic detachment of instances is that when you close your

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

PersistenceManager you can opt to have all instances currently cached in the Level 1 Cache of that PersistenceManager detached automatically. This means that you can persist instances, and then when you close the PM the instances will be detached and ready for further work. This is a DataNucleus extension. **It is recommended that you use "detachAllOnCommit" since that is standard JDO and since this option will not work in J2EE environments where the PersistenceManager close is controlled by the J2EE container**

You enable this by setting the PersistenceManagerFactory (PMF) property datanucleus.DetachOnClose when you create the PMF. Let's give an example

```
// Create a PMF with the datanucleus.DetachOnClose property set to "true"
...

// Create an object
MyObject my = new MyObject();

PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    // Persist the object that we created earlier
    pm.makePersistent(my);

    tx.commit();

    pm.close();
    // The object "my" is now in detached state and can be used further
}
finally
{
    if (tx.isActive)
    {
        tx.rollback();
    }
}
```

That is about as close to transparent persistence as you will find. When the PM is closed all instances found in the L1 Cache are detached using the current FetchPlan, and so all fields in that plan for the instances in question will be detached at that time.

Detached Fields

When an object is detached it is typically passed to a different layer of an application and potentially changed. During the course of the operation of the system it may be required to know what is loaded in the object and what is dirty (has been changed since detaching). DataNucleus provides an extension to allow interrogation of the detached object.

```
String[] loadedFieldNames = NucleusJDOHelper.getDetachedObjectLoadedFields(obj, pm);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
String[] dirtyFieldNames = NucleusJDOHelper.getDetachedObjectDirtyFields(obj, pm);
```

So you have access to the names of the fields that were loaded when detaching the object, and also to the names of the fields that have been updated since detaching.

Serialization of Detachable classes

During enhancement of Detachable classes, a field called *jdoDetachedState* is added to the class definition. This field allows reading and changing tracking of detached objects while they are not managed by a PersistenceManager.

When serialization occurs on a Detachable object, the *jdoDetachedState* field is written to the serialized object stream. On deserialize, this field is written back to the new deserialized instance. This process occurs transparently to the application. However, if deserialization occurs with an un-enhanced version of the class, the detached state is lost.

Serialization and deserialization of Detachable classes and un-enhanced versions of the same class is only possible if the field *serialVersionUID* is added. It's recommended during development of the class, to define the *serialVersionUID* and make the class to implement the *java.io.Serializable* interface, as the following example:

```
class MyClass implements java.io.Serializable
{
    private static final long serialVersionUID = 2765740961462495537L; // any random
value here

    //.... other fields
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.13 Fetch Groups

Fetch Groups

When an object is retrieved from the datastore by JDO typically not all fields are retrieved immediately. This is because for efficiency purposes only particular field types are retrieved in the initial access of the object, and then any other objects are retrieved when accessed (lazy loading). The group of fields that are loaded is called a fetch group. There are 3 types of "fetch groups" to consider

- Default Fetch Group : defined in all JDO specs, containing the fields of a class that will be retrieved by default (with no user specification).
- Named Fetch Groups : defined by the JDO2 specification, and defined in MetaData (XML/annotations) with the fields of a class that are part of that fetch group. The definition here is *static*
- Dynamic Fetch Groups : Programmatic definition of fetch groups at runtime via an API

The fetch group in use for a class is controled via the FetchPlan Javadoc interface. To get a handle on the current FetchPlan we do

```
FetchPlan fp = pm.getFetchPlan();
```

Default Fetch Group

JDO provides an initial fetch group, comprising the fields that will be retrieved when an object is retrieved if the user does nothing to define the required behaviour. By default the default fetch group comprises all fields of the following types :-

- primitives : boolean, byte, char, double, float, int, long, short
- Object wrappers of primitives : Boolean, Byte, Character, Double, Float, Integer, Long, Short
- java.lang.String, java.lang.Number, java.lang.Enum
- java.math.BigDecimal, java.math.BigInteger
- java.util.Date

If you wish to change the Default Fetch Group for a class you can update the Meta-Data for the class as follows (for XML)

```
<class name="MyClass">
    ...
    <field name="fieldX" default-fetch-group="true"/>
</class>
```

or using annotations

 ALL RIGHTS RESERVED

```
@Persistent(defaultFetchGroup="true")
SomeType fieldX;
```

When a PersistenceManager is created it starts with a FetchPlan of the "default" fetch group. That is, if we call

```
Collection fetchGroups = fp.getGroups();
```

this will have one group, called "default". At runtime, if you have been using other fetch groups and want to revert back to the default fetch group at any time you simply do

```
fp.setGroup(FetchPlan.DEFAULT);
```

Named Fetch Groups

As mentioned above, JDO2 allows specification of users own fetch groups. These are specified in the MetaData of the class. For example, if we have the following class

```
class MyClass
{
    String name;
    HashSet coll;
    MyOtherClass other;
}
```

and we want to have the other field loaded whenever we load objects of this class, we define our MetaData as

```
<package name="mydomain">
    <class name="MyClass">
        <field name="name">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="coll" persistence-modifier="persistent">
            <collection element-type="mydomain.Address"/>
            <join/>
        </field>
        <field name="other" persistence-modifier="persistent"/>
        <fetch-group name="otherfield">
            <field name="other"/>
        </fetch-group>
    </class>
</package>
```

So we have defined a fetch group called "otherfield" that just includes the field with name other. We can

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

then use this at runtime in our persistence code.

```
PersistenceManager pm = pmf.getPersistenceManager();
pm.getFetchPlan().addGroup("otherfield");

... (load MyClass object)
```

By default the FetchPlan will include the default fetch group. We have changed this above by adding the fetch group "otherfield", so when we retrieve an object using this PersistenceManager we will be retrieving the fields name AND other since they are both in the current FetchPlan. We can take the above much further than what is shown by defining nested fetch groups in the MetaData. In addition we can change the FetchPlan just before any PersistenceManager operation to control what is fetched during that operation. The user has full flexibility to add many groups to the current Fetch Plan. This gives much power and control over what will be loaded and when. A big improvement over JDO 1.0

The FetchPlan applies not just to calls to PersistenceManager.getObjectById(), but also to PersistenceManager.newQuery(), PersistenceManager.getExtent(), PersistenceManager.detachCopy and much more besides.

To read more about named fetch-groups and how to use it with attach/detach you can look at our Tutorial on DAO Layer design .

Dynamic Fetch Groups

The mechanism above provides static fetch groups defined in XML or annotations. That is great when you know in advance what fields you want to fetch. In some situations you may want to define your fields to fetch at run time. This became standard in JDO2.2 (was previously a DataNucleus extension). It operates as follows

```
import org.datanucleus.FetchGroup;

// Create a FetchGroup on the PMF called "TestGroup" for MyClass
FetchGroup grp = myPMF.getFetchGroup("TestGroup", MyClass.class);
grp.addMember("field1").addMember("field2");

// Add this group to the fetch plan (using its name)
fp.addGroup("TestGroup");
```

So we use the DataNucleus PMF as a way of creating a FetchGroup, and then register that FetchGroup with the PMF for use by all PMs. We then enable our FetchGroup for use in the FetchPlan by using its group name (as we do for a static group). The FetchGroup allows you to add/remove the fields necessary so you have full API control over the fields to be fetched.

Fetch Depth

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The basic fetch group defines which fields are to be fetched. It doesn't explicitly define how far down an object graph is to be fetched. JDO2 provides two ways of controlling this.

The first is to set the maxFetchDepth for the FetchPlan. This value specifies how far out from the root object the related objects will be fetched. A positive value means that this number of relationships will be traversed from the root object. A value of -1 means that no limit will be placed on the fetching traversal. The default is 1. Let's take an example

```
public class MyClass1
{
    MyClass2 field1;
    ...
}

public class MyClass2
{
    MyClass3 field2;
    ...
}

public class MyClass3
{
    MyClass4 field3;
    ...
}
```

and we want to detach field1 of instances of MyClass1, down 2 levels - so detaching the initial "field1" MyClass2 object, and its "field2" MyClass3 instance. So we define our fetch-groups like this

```
<class name="MyClass1">
    ...
    <fetch-group name="includingField1">
        <field name="field1"/>
    </fetch-group>
</class>
<class name="MyClass2">
    ...
    <fetch-group name="includingField2">
        <field name="field2"/>
    </fetch-group>
</class>
```

and we then define the maxFetchDepth as 2, like this

```
pm.getFetchPlan().setMaxFetchDepth(2);
```

A further refinement to this global fetch depth setting is to control the fetching of recursive fields. This is performed via a MetaData setting "recursion-depth". A value of 1 means that only 1 level of objects will be fetched. A value of -1 means there is no limit on the amount of recursion. The default is 1. Let's take an example

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class Directory
{
    Collection children;
    ...
}
```

```
<class name="Directory">
    <field name="children">
        <collection element-type="Directory"/>
    </field>

    <fetch-group name="grandchildren">
        <field name="children" recursion-depth="2"/>
    </fetch-group>
    ...
</class>
```

So when we fetch a Directory, it will fetch 2 levels of the children field, hence fetching the children and the grandchildren.

Fetch Size

A FetchPlan can also be used for defining the fetching policy when using queries. This can be set using

```
pm.getFetchPlan().setFetchSize(value);
```

The default is FetchPlan.FETCH_SIZE_OPTIMAL which leaves it to DataNucleus to optimise the fetching of instances. A positive value defines the number of instances to be fetched. Using FetchPlan.FETCH_SIZE_GREEDY means that all instances will be fetched immediately.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.14 Instance Callbacks

JDO Instance Callbacks

JDO 1.0 and 2.0 define an interface for PersistenceCapable classes so that they can be notified of events in their own lifecycle and perform any additional operations that are needed at these checkpoints. This is a complement to the Lifecycle Listeners interface which provides listeners for all objects of particular classes, with the events sent to a listener. With InstanceCallbacks the PersistenceCapable class is the destination of the lifecycle events. As a result the Instance Callbacks method is more intrusive than the method of Lifecycle Listeners in that it requires methods adding to each class that wishes to receive the callbacks.

DataNucleus supports the InstanceCallbacks interface Javadoc.

To give an example of this capability, let us define a class that needs to perform some operation just before it's object is deleted.

```
public class MyClass implements InstanceCallbacks
{
    String name;

    ... (class methods)


    public void jdoPostLoad() {}
    public void jdoPreClear() {}
    public void jdoPreStore() {}

    public void jdoPreDelete()
    {
        // Perform some operation just before being deleted.
    }
}
```

So we have implemented InstanceCallbacks and have defined the 4 required methods. Only one of these is of importance in this example.

These methods will be called just before storage in the data store (jdoPreStore), just before clearing (jdoPreClear), just after being loaded from the datastore (jdoPostLoad) and just before being deleted (jdoPreDelete).

JDO2 adds 2 new callbacks to complement InstanceCallbacks. These are AttachCallback Javadoc and DetachCallback Javadoc. If you want to intercept attach/detach events your class can implement these interfaces. You will then need to implement the following methods

```
public interface AttachCallback
{
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    public void jdoPreAttach();
    public void jdoPostAttach(Object attached);
}

public interface DetachCallback
{
    public void jdoPreDetach();
    public void jdoPostDetach(Object detached);
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.15 Lifecycle Listeners

Instance Lifecycle Listeners

JDO 2.0 defines an interface for the PersistenceManager and PersistenceManagerFactory whereby a user can register a listener for persistence events. The user provides a listener for either all classes, or a set of defined classes, and the JDO implementation calls methods on the listener when the required events occur. This provides the user application with the power to monitor the persistence process and, where necessary, append related behaviour. Specifying the listeners on the PersistenceManagerFactory has the benefits that these listeners will be added to all PersistenceManagers created by that factory, and so is for convenience really. This facility is a complement to the Instance Callbacks facility which allows interception of events on an instance by instance basis. The Lifecycle Listener process is much less intrusive than the process provided by Instance Callbacks, allowing a class external to the persistence process to perform the listening.

DataNucleus supports the InstanceLifecycleListener interface. .

To give an example of this capability, let us define a Listener for our persistence process.

```
public class LoggingLifecycleListener implements CreateLifecycleListener,
    DeleteLifecycleListener, LoadLifecycleListener, StoreLifecycleListener
{
    public void postCreate(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : create for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void preDelete(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : preDelete for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void postDelete(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : postDelete for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void postLoad(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : load for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void preStore(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : preStore for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    public void postStore(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : postStore for " +
            ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

}
```

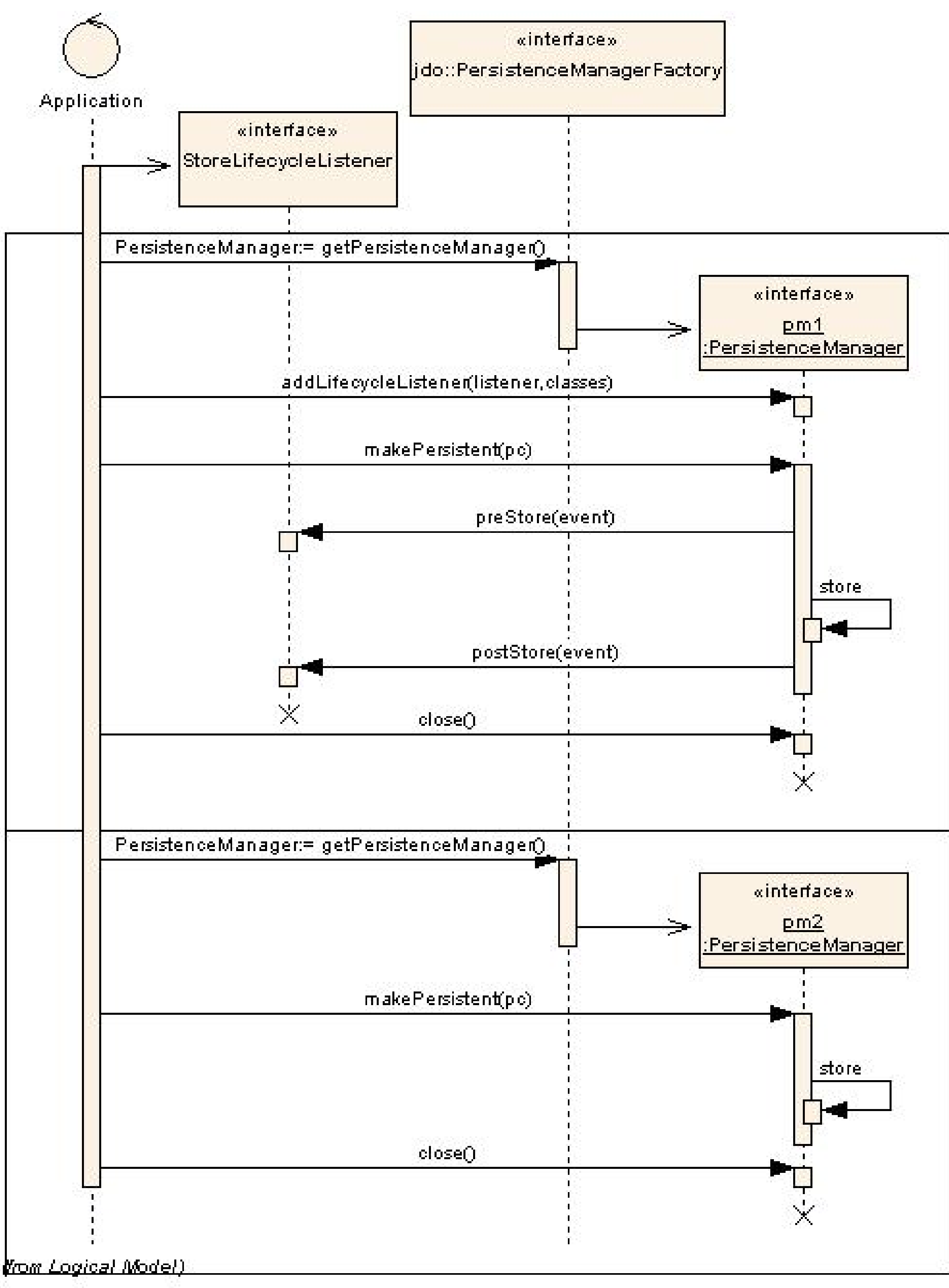
Here we've provided a listener to receive events for CREATE, DELETE, LOAD, and STORE of objects. These are the main event types and in our simple case above we will simply log the event. All that remains is for us to register this listener with the PersistenceManager, or PersistenceManagerFactory

```
pm.addInstanceLifecycleListener(new LoggingLifecycleListener(), null);
```

When using this interface the user should always remember that the listener is called within the same transaction as the operation being reported and so any changes they then make to the objects in question will be reflected in that objects state.

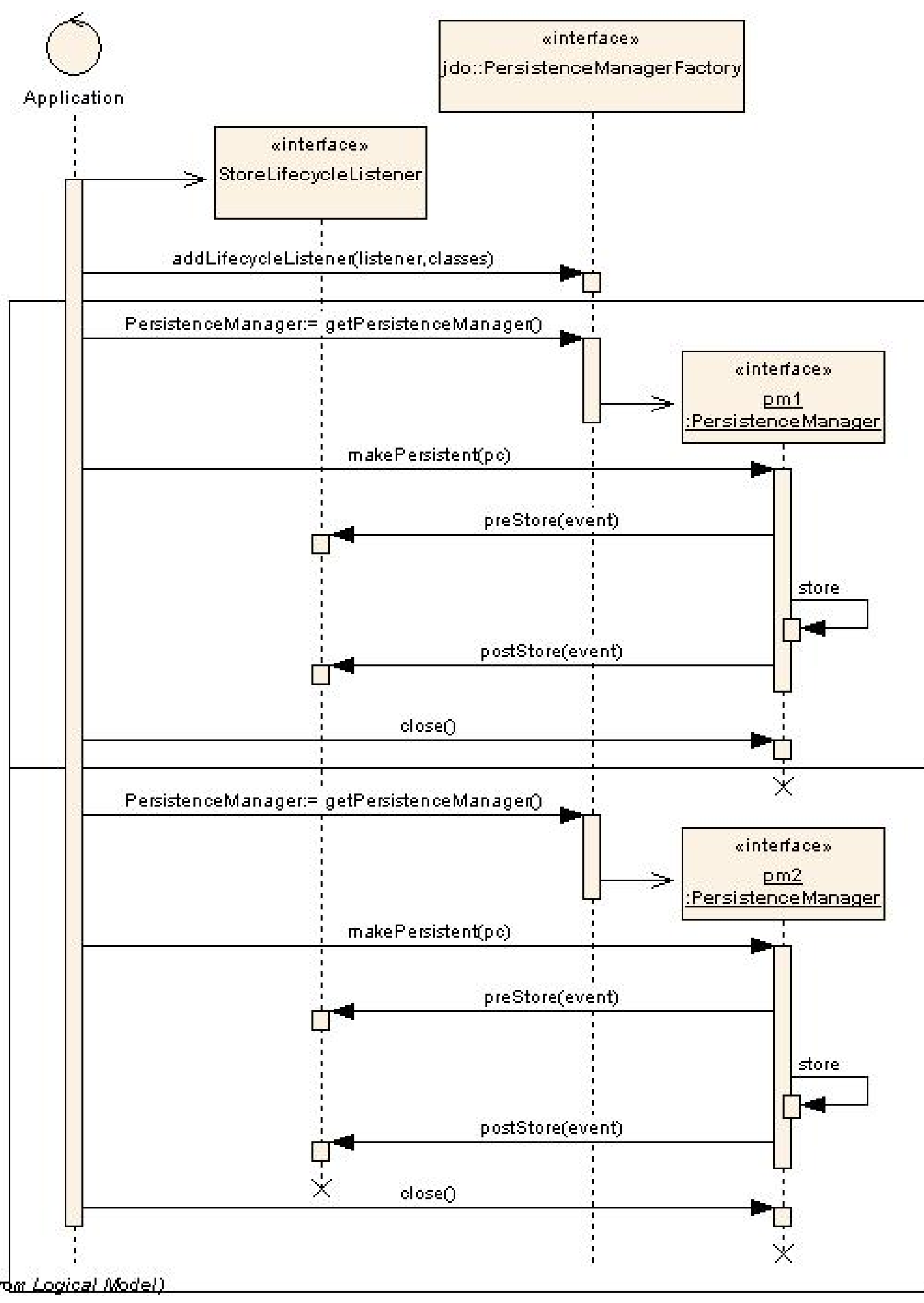
Register the listener with the PersistenceManager or PersistenceManagerFactory provide different effects. Registering with the PersistenceManagerFactory means that all PersistenceManagers created by it will have the listeners registered on the PersistenceManagerFactory called. Registering the listener with the PersistenceManager will only have the listener called only on events raised only by the PersistenceManager instance.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The above diagram displays the sequence of actions for a listener registered only in the PersistenceManager. Note that a second PersistenceManager will not make calls to the listener registered in the first PersistenceManager.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The above diagram displays the sequence of actions for a listener registered in the PersistenceManagerFactory. All events raised in a PersistenceManager obtained from the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

PersistenceManagerFactory will make calls to the listener registered in the PersistenceManagerFactory.

Available Listener Types

DataNucleus supports the following instance lifecycle listener types

- AttachLifecycleListener - all attach events
- ClearLifecycleListener - all clear events
- CreateLifecycelListener - all object create events
- DeleteLifecycleListener - all object delete events
- DetachLifecycleListener - all detach events
- DirtyLifecycleListener - all dirty events
- LoadLifecycleListener - all load events
- StoreLifecycleListener - all store events

Fields being stored

The default JDO2 lifecycle listener *StoreLifecycleListener* only informs the listener of the object being stored. It doesn't provide information about the fields being stored in that event. DataNucleus extends the JDO2 specification and on the "preStore" event it will return an instance of *org.datanucleus.FieldInstanceLifecycleEvent* (which extends the JDO2 InstanceLifecycleEvent) and provides access to the names of the fields being stored.

```
public class FieldInstanceLifecycleEvent extends InstanceLifecycleEvent
{
    ...

    /**
     * Accessor for the field names affected by this event
     * @return The field names
     */
    public String[] getFieldNames()
    ...
}
```

If the store event is the persistence of the object then this will return all field names. If instead just particular fields are being stored then you just receive those fields in the event. So the only thing to do to utilise this DataNucleus extension is cast the received event to *org.datanucleus.FieldInstanceLifecycleEvent*

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

9.16 J2EE and JDO

DataNucleus - Usage within a J2EE environment

The J2EE framework has become popular in some places in the last few years. It provides a container within which java processes operate and it provides mechanisms for, amongst other things, transactions (JTA), and for connecting to other (3rd party) utilities (using Java Connector Architecture, JCA). DataNucleus Access Platform can be utilised within a J2EE environment via this JCA system, and we provide a Resource Adaptor (RAR file) containing this JCA adaptor allowing Access Platform to be used with the likes of WebLogic and JBoss. Instructions are provided for the following J2EE servers

- WebLogic
- JBoss 3.0/3.2
- JBoss 4.0
- Jonas 4.8

The provided DataNucleus JCA rar provides default resource adapter descriptors, one general, and the other for the WebLogic J2EE server. These resource adapter descriptors can be configured to meet your needs, for example allowing XA transactions instead of the default Local transactions.

Requirements

To use DataNucleus with JCA the first thing that you will require is the datanucleus-jca-{version}.rar file (available from the download section).

DataNucleus Resource Adaptor and transactions

A great advantage of DataNucleus implementing the ManagedConnection interface is that the J2EE container manages transactions for you (no need to call the begin/commit/rollback-methods). Currently, local transactions and distributed (XA) transactions are supported. Within a J2EE environment, JDO transactions are nested in J2EE transactions. All you have to do is to declare that a method needs transaction management. This is done in the EJB meta data. Here you will see, how a SessionBean implementation could look like.

The EJB meta data is defined in a file called ejb-jar.xml and can be found in the META-INF directory of the jar you deploy. Suppose you deploy a bean called DataNucleusBean, your ejb-jar.xml should contain the following configuration elements:

```
<session>
        <ejb-name>DataNucleusBean</ejb-name>
        ...
        <transaction-type>Container</transaction-type>
        ...
<session>
```

Imagine your bean defines a method called testDataNucleusTrans():

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<container-transaction>
    <method >
        <ejb-name>DataNucleusBean</ejb-name>
        ...
        <method-name>testDataNucleusTrans</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
```

You hereby define that transaction management is required for this method. The container will automatically begin a transaction for this method. It will be commited if no error occurs or rolled back otherwise. A potential SessionBean implementation containing methods to retrieve a PersistenceManager then could look like this:

```
public abstract class DataNucleusBean implements SessionBean
{
    // EJB methods
    public void ejbCreate()
    throws CreateException
    {
    }

    public void ejbRemove()
    throws EJBException, RemoteException
    {
    }

    // convenience methods to get
    // a PersistenceManager

    /**
     * static final for the JNDI name of the PersistenceManagerFactory
     */
    private static final String PMF_JNDI_NAME = "java:/datanucleus1";

    /**
     * Method to get the current InitialContext
     */
    private InitialContext getInitialContext() throws NamingException
    {
        InitialContext initialContext = new InitialContext();
        // or other code to create the InitialContext eg. new
InitialContext(myProperies);
        return initialContext;
    }

    /**
     * Method to lookup the PersistenceManagerFactory
     */
    private PersistenceManagerFactory getPersitenceManagerFactory(InitialContext
context)
    throws NamingException
    {
        return (PersistenceManagerFactory) context.lookup(PMF_JNDI_NAME);
    }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Method to get a PersistenceManager
     */
    public PersistenceManager getPersistenceManager()
    throws NamingException
    {
        return
getPersitenceManagerFactory(getInitialContext()).getPersistenceManager();
    }

    // Now finally the bean method within a transaction

    public viod testDataNucleusTrans()
    throws Exception
    {
        PersistenceManager pm = getPersistenceManager()
        try {
            // Do something with your PersistenceManager
        } finally {
            // close the PersistenceManager
            pm.close();
        }
    }
}
```

Make sure, you close the PersistenceManager in your bean methods. If you don't, the J2EE server will usually close it for you (one of the advantages), but of course not without a warning or error message.

To avoid the need of editing multiple files, you could use XDoclet to generate your classes and control the metadata by xdoclet tags. The method declaration then would look like this:

```
    /**
     * @ejb.interface-method
     * @ejb.transaction type="Required"
     */
    public viod testDataNucleusTrans()
    throws Exception
    {
        //...
    }
```

These instructions were adapted from a contribution by a DataNucleus user Alexander Bieber.

Persistence Properties

When creating a PMF using the JCA adaptor, you should specify your persistence properties using a persistence.xml or jdoconfig.xml. This is because DataNucleus JCA adapter from version 1.2.2 does not support Java bean setters/getters for all properties - since it is an inefficient and inflexible mechanism for property specification. The more recent *persistence.xml* and *jdoconfig.xml* methods lead to more extensible code.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

General configuration

A resource adapter has one central configuration file /META-INF/ra.xml which is located within the rar file and which defines the default values for all instances of the resource adapter (i.e. all instances of *PersistenceManagerFactory*). Additionally, it uses one or more deployment descriptor files (in JBoss, for example, they are named _*-ds.xml_) to set up the instances. In these files you can override the default values from the *ra.xml*.

Since it is bad practice (and inconvenient) to edit a library's archive (in this case the *datanucleus-jca-${version}.rar*) for changing the configuration (it makes updates more complicated, for example), it is recommended, not to edit the *ra.xml* within DataNucleus' rar file, but instead put all your configuration into your deployment descriptors. This way, you have a clean separation of which files you maintain (your deployment descriptors) and which files are maintained by others (the libraries you use and which you simply replace in case of an update).

Nevertheless, you might prefer to declare default values in the *ra.xml* in certain circumstances, so here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connector PUBLIC "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
    "http://java.sun.com/dtd/connector_1_0.dtd">
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
<managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnec
<connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
<connectionfactory-impl-class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfact
        <connection-interface>javax.resource.cci.Connection</connection-interface>
<connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>LocalTransaction</transaction-support>
        <config-property>
          <config-property-name>ConnectionFactoryName</config-property-name>
          <config-property-type>java.lang.String</config-property-type>
          <config-property-value>jdbc/ds</config-property-value>
        </config-property>
        <authentication-mechanism>
<authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
<credential-interface>javax.resource.security.PasswordCredential</credential-interface>
        </authentication-mechanism>
        <reauthentication-support>false</reauthentication-support>
    </resourceadapter>
</connector>
```

To define persistence properties you should make use of **persistence.xml** or **jdoconfig.xml** and refer to the documentation for persistence properties for full details of the properties.

WebLogic

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

To use DataNucleus on Weblogic the first thing that you will require is the datanucleus-{version}.rar file. You then may need to edit the /META-INF/weblogic-ra.xml file to suit the exact version of your WebLogic server (the included file is for WebLogic 8.1).

You then deploy the RAR file on your WebLogic server.

JBoss 3.0/3.2

To use DataNucleus on JBoss (Ver 3.2) the first thing that you will require is the datanucleus-{version}.rar file. You should put this in the deploy ("${JBOSS}/server/default/deploy/") directory of your JBoss installation.

You then create a file, also in the deploy directory with name datanucleus-ds.xml. To give a guide on what this file will typically include, see the following

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
    <tx-connection-factory>
        <jndi-name>datanucleus</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
type="java.lang.String">jdbc:mysql://localhost/yourdbname</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>

    <tx-connection-factory>
        <jndi-name>datanucleus1</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
type="java.lang.String">jdbc:mysql://localhost/yourdbname1</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>

    <tx-connection-factory>
        <jndi-name>datanucleus2</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
type="java.lang.String">jdbc:mysql://localhost/yourdbname2</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>
</connection-factories>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This example creates 3 connection factories to MySQL databases, but you can create as many or as few as you require for your system to whichever databases you prefer (as long as they are supported by DataNucleus). With the above definition we can then use the JNDI names java:/datanucleus, java:/datanucleus1, and java:/datanucleus2 to refer to our datastores.

Note, that you can use separate deployment descriptor files. That means, you could for example create the three files *datanucleus1-ds.xml*, *datanucleus2-ds.xml* and *datanucleus3-ds.xml* with each declaring one *PersistenceManagerFactory* instance. This is useful (or even required) if you need a distributed configuration. In this case, you can use JBoss' hot deployment feature and deploy a new *PersistenceManagerFactory*, while the server is running (and working with the existing PMFs): If you create a new **-ds.xml* file (instead of modifying an existing one), the server does not undeploy anything (and thus not interrupt ongoing work), but will only add the new connection factory to the JNDI.

You are now set to work on DataNucleus-enabling your actual application. As we have said, you can use the above JNDI names to refer to the datastores, so you could do something like the following to access the PersistenceManagerFactory to one of your databases.

```
import javax.jdo.PersistenceManagerFactory;

InitialContext context = new InitialContext();
PersistenceManagerFactory pmf =
(PersistenceManagerFactory)context.lookup("java:/datanucleus1");
```

These instructions were adapted from a contribution by a DataNucleus user Marco Schulze.

JBoss 4.0

With JBoss 4.0 there are some changes in configuration relative to JBoss 3.2 in order to allow use some new features of JCA 1.5. Here you will see how to configure JBoss 4.0 to use with DataNucleus JCA adapter for Apache Derby DBMS.

To use DataNucleus on JBoss 4.0 the first thing that you will require is the datanucleus-{version}.rar file. You should put this in the deploy directory ("${JBOSS}/server/default/deploy/") of your JBoss installation. Additionally, you have to remember to put any JDBC driver files to lib directory ("${JBOSS}/server/default/lib/") if JBoss does not have them installed by default. In case of Apache Derby you need to copy db2jcc.jar and db2jcc_license_c.jar.

You then create a file, also in the deploy directory with name datanucleus-ds.xml. To give a guide on what this file will typically include, see the following

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
    <tx-connection-factory>
        <jndi-name>datanucleus</jndi-name>
        <rar-name>datanucleus-{version}.rar</rar-name> <!-- the name here must be
the same as JCA adapter filename -->
<connection-definition>javax.resource.cci.ConnectionFactory</connection-definition>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.ibm.db2.jcc.DB2Driver</config-property>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <config-property name="ConnectionURL"
type="java.lang.String">jdbc:derby:net://localhost:1527/"directory_of_your_db_files"</config-property>
        <config-property name="UserName"
            type="java.lang.String">app</config-property>
        <config-property name="Password"
            type="java.lang.String">app</config-property>
        </tx-connection-factory>
</connection-factories>
```

You are now set to work on DataNucleus-enabling your actual application. You can use the above JNDI name to refer to the datastores, and so you could do something like the following to access the PersistenceManagerFactory to one of your databases.

```
import javax.jdo.PersistenceManagerFactory;

InitialContext context=new InitialContext();
PersistenceManagerFactory
pmFactory=(PersistenceManagerFactory)context.lookup("java:/datanucleus");
```

These instructions were adapted from a contribution by a DataNucleus user Maciej Wegorkiewicz.

Jonas

To use DataNucleus on Jonas the first thing that you will require is the datanucleus-{version}.rar file. You then may need to edit the /META-INF/jonas-ra.xml file to suit the exact version of your Jonas server (the included file is tested for Jonas 4.8).

You then deploy the RAR file on your Jonas server.

Transaction Support

DataNucleus JCA adapter supports both Local and XA transaction types. Local means that a transaction will not have more than one resource managed by a Transaction Manager and XA means that multiple resources are managed by the Transaction Manager. Use XA transaction, if DataNucleus is configured to use data sources deployed in application servers, or if other resources such as JMS connections are used in the same transaction, otherwise use Local transaction.

You need to configure the *ra.xml* file with the appropriate transaction support, which is either *XATransaction* or *LocalTransaction*. See the example:

```
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
<managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconne
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
<connectionfactory-impl-class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfact
        <connection-interface>javax.resource.cci.Connection</connection-interface>
<connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>XATransaction</transaction-support> <!-- change this
line -->
    ...
```

Data Source

To use a data source, you have to configure the connection factory name in *ra.xml* file. See the example:

```
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
<managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnec
<connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
<connectionfactory-impl-class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfact
        <connection-interface>javax.resource.cci.Connection</connection-interface>
<connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>XATransaction</transaction-support>

        <config-property>
            <config-property-name>ConnectionFactoryName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>jndiName_for_datasource_1</config-property-value>
        </config-property>
            <config-property>
            <config-property-name>ConnectionResourceType</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>JTA</config-property-value>
        </config-property>
        <config-property>
            <config-property-name>ConnectionFactory2Name</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>jndiName_for_datasource_2</config-property-value>
        </config-property>
    ...
```

See also :

- RDBMS Data Sources usage with DataNucleus

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.1 JDO Query API

JDO Query API

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JDO specifies support for an object-oriented query language (JDOQL) and a relational query language (SQL). DataNucleus supports the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- SQL - language found on alomst all RDBMS. This is clearly of little use when using an object datastore for persistence
- JPQL - language defined in the JPA1 specification for JPA persistence but provided here so that DataNucleus users have full flexibility to mix-and-match specifications

Note that depending on the datastore in use there may be alternative query languages available. Please check the docs for the datastore.

Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer JDOQL. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of SQL. This is the power of an implementation like DataNucleus in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

There are 2 categories of queries with JDO :-

- Named Query where the query is defined in MetaData and referred to by its name at runtime.
- **Programmatic Query** where the query is defined using the JDO2 Query API.

Let's now try to understand the Query API in JDO Javadoc, We firstly need to look at a typical Query.

```
Query query = pm.newQuery("javax.jdo.query.JDOQL", "SELECT FROM
org.datanucleus.MyClass WHERE param2 < threshold");
query.declareImports("import java.util.Date");
query.declareParameters("Date threshold");
query.setOrdering("param1 ascending");
List results = (List)query.execute(my_threshold);
```

In this Query, we select our query language (JDOQL in this case), and the query is specified to return all objects of type org.datanucleus.MyClass (or subclasses) which have the field param2 less than some threshold value. We've specified the query like this because we want to pass the threshold value in dynamically. We then import the type of our threshold parameter, and the parameter itself, and set the ordering of the results from the Query to be in ascending order of some field param1. The Query is then executed, passing in the threshold value. The example is to highlight the typical methods specified for a Query. Clearly you may only specify the Query line if you wanted something very simple. The result of the Query is cast to a List since in this case it returns a List of results.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Let's review the methods of the JDO Query API. Not all methods apply to all query languages. The differences are noted in the sections below

setClass()

Set the class of the candidate instances of the query. The class specifies the class of the candidates of the query. Elements of the candidate collection that are of the specified class are filtered before being put into the results.

This is applicable to JDOQL, SQL, JPQL.

setUnique()

Specify that only the first result of the query should be returned, rather than a collection. The execute method will return null if the query result size is 0.

This is applicable to JDOQL, SQL, JPQL.

Sometimes you know that the query can only every return 0 or 1 objects. In this case you can simplify your job by adding

```
query.setUnique(true);
```

In this case the return from the execution of the Query will be a single Object, so you've no need to use iterators, just cast it to your candidate class type.

setResult()

Specifies what type of data this query should return. If this is unset or set to null, this query returns instances of the query's candidate class. If set, this query will return expressions, including field values (projections) and aggregate function results.

This is applicable to JDOQL.

The normal behaviour of JDOQL queries is to return a List of Objects of the type of the candidate class. Sometimes you want to have the query perform some processing and return things like count(), min(), max() etc. You specify this with

```
query.setResult("count(param1), max(param2), param3");
```

In this case the results will be List<Object[]> since there are more than 1 column in each row. If you have only 1 column in the results then the results would be List<Object>. If you have only aggregates (sum, avg, min, max, count) in the result clause then there will be only 1 row in the results and so the results will be of the form Object[] (or Object if only 1 aggregate).

setResultClass()

Specify the type of object in which to return each element of the result of invoking execute(). If the result is not set or set to null, the result class defaults to the candidate class of the query. If the result consists of one expression, the result class

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

defaults to the type of that expression. If the result consists of more than one expression, the result class defaults to Object[].

This is applicable to JDOQL, SQL.

When you perform a query, using JDOQL or SQL the query will, in general, return a List of objects. These objects are by default of the same type as the candidate class. This is good for the majority of situations but there are some situations where you would like to control the output object. This can be achieved by specifying the Result Class.

```
query.setResultClass(myResultClass);
```

The Result Class has to meet certain requirements. These are

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

Where you have a query returning a single field, you could specify the Result Class to be one of the first group for example. Where your query returns multiple fields then you can set the Result Class to be your own class. So we could have a query like this

```
Query query =
pm.newQuery(pm.getExtent(org.datanucleus.samples.Payment.class,false));
query.setFilter("amount > 10.0");
query.setResultClass(Price.class);
query.setResult("amount, currency");
List results = (List)query.execute();
```

and we define our Result Class Price as follows

```
public class Price
{
    protected double amount = 0.0;
    protected String currency = null;

    public Price(double amount, String currency)
    {
        this.amount = amount;
        this.currency = currency;
    }

    ...
}
```

In this case our query is returning 2 fields (a Double and a String), and these map onto the constructor arguments, so DataNucleus will create objects of the Price class using that constructor. We could have

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

provided a class with public fields instead, or provided setXXX methods or a put method. They all work in the same way.

setRange()

Set the range of results to return. The execution of the query is modified to return only a subset of results. If the filter would normally return 100 instances, and fromIncl is set to 50, and toExcl is set to 70, then the first 50 results that would have been returned are skipped, the next 20 results are returned and the remaining 30 results are ignored. An implementation should execute the query such that the range algorithm is done at the data store.

This is applicable to JDOQL.

Sometimes you have a Query that returns a large number of objects. You may want to just display a range of these to your user. In this case you can do

```
query.setRange(10, 20);
```

This has the effect of only returning items 10 through to 19 (inclusive) of the query's results. The clear use of this is where you have a web system and you're displaying paginated data, and so the user hits page down, so you get the next "n" results.

setRange is implemented efficiently for MySQL, Postgresql, HSQL (using the LIMIT SQL keyword) and Oracle (using the ROWNUM keyword), with the query only finding the objects required by the user directly in the datastore. For other RDBMS the query will retrieve all objects up to the "to" record, and will not pass any unnecessary objects that are before the "from" record.

setFilter()

Set the filter for the query. The filter specification is a String containing a Boolean expression that is to be evaluated for each of the instances in the candidate collection. If the filter is not specified, then it defaults to "true", which has the effect of filtering the input `Collection` only for class type.

This is applicable to JDOQL.

declareImports()

Set the import statements to be used to identify the fully qualified name of variables or parameters. Parameters and unbound variables might come from a different class from the candidate class, and the names need to be declared in an import statement to eliminate ambiguity. Import statements are specified as a String with semicolon-separated statements.

This is applicable to JDOQL.

In JDOQL you can declare parameters and variables. Just like in Java it is often convenient to just declare a variable as say Date, and then have an import in your Java file importing the java.util.Date class. The same applies in JDOQL. Where you have defined parameters or variables in shorthand form, you can specify their imports like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
query.declareVariables("Date startDate");
query.declareParameters("Locale myLocale");
query.declareImports("import java.util.Locale; import java.util.Date;");
```

Just like in Java, if you declare your parameters or variables in fully-specified form (for example "java.util.Date myDate") then you do not need any import.

The JDOQL uses the *imports* declaration to create a *type namespace* for the query. During query compilation, the classes used in the query, if not fully qualified, are searched in this namespace. The type namespace is built with the following:

- primitives types
- java.lang.* package
- package of the candidate class
- import declarations (if any)

To resolve a class, the JDOQL compiler will use the class fully qualified name to load it, but if the class is not fully qualified, it will search by prefixing the class name with the imported package names declared in the type namespace. All classes loaded by the query must be acessible by either the candidate class classloader, the PersistenceManager classloader or the current Thread classloader. The search algorithm for a class in the JDOQL compiler is the following:

- if the class is fully qualified, load the class.
- if the class is not fully qualified, iterate each package in the type namespace and try to load the class from that package. This is done until the class is loaded, or the type namespace package names are exausted. If the class cannot be loaded an exception is thrown.

Note that the search algorithm can be problematic in performance terms if the class is not fully qualified or declared in imports using package notation. To avoid such problems, either use fully qualified class names or import the class in the imports declaration. The 2 queries below are examples of good usage:

```
query.declareImports("import java.util.Locale;");
query.declareParameters("Locale myLocale");
```

or

```
query.declareParameters("java.util.Locale myLocale");
```

However, the below example will suffer in performance, due to the search algorithm.

```
query.declareImports("import java.math.*; import java.util.*;");
query.declareParameters("Locale myLocale");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

declareParameters()

Declare the list of parameters query execution. The parameter declaration is a String containing one or more query parameter declarations separated with commas. Each parameter named in the parameter declaration must be bound to a value when the query is executed.

This is applicable to JDOQL.

When specifying JDOQL queries you can use normal Java classes in the query. You need to define to JDOQL which class you are talking about. You can do this via the following calls

```
query.declareImports("import mypackage.myclass; import mypackage2.*");
query.declareParameters("String myparam1, Date myparam2");
```

This tells JDO (and more specifically DataNucleus) to look for any classes specified as parameters, in the defined list of imports. java.lang is imported automatically here so you don't need to specify this. It should be noted that if you specify a package in the imports (using * notation), then DataNucleus will have to search for your class in that package and this can have a (small) performance impact. You can get around this by not using "*" notation on imports. What are shown above are called explicit parameters.

In JDO2 you can alternatively utilise implicit parameters. This is done by specifying the parameter name in the query directly (prefixed by a colon). So you may write a Single-String query like

```
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE
price < :limit");
List results = (List)query.execute(new Double(200.0));
```

Here we haven't declared the parameter. We simply prefixed it with a colon, and its type will be determined implicitly from the query.

In some situations you may have a map of parameters keyed by their name, yet the query in question doesn't need all parameters. Normal JDO execution would throw an exception here since they are inconsistent with the query. You can omit this check by setting

```
q.addExtension("datanucleus.query.ignoreParameterCountCheck", "true");
```

declareVariables()

Declare the unbound variables to be used in the query. Variables might be used in the filter, and these variables must be declared with their type. The unbound variable declaration is a String containing one or more unbound variable declarations separated with semicolons.

This is applicable to JDOQL.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

In JDOQL you can connect two parts of a query using something known as a variable. For example, we want to retrieve all objects with a collection that contains a particular element, and where the element has a particular field value. We define a query like this

```
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Supplier WHERE
products.contains(prod) &&
                    prod.name == \"Beans\"");
```

So we have a variable in our query called "prod" that connects the two parts. We now declare it

```
query.declareVariables("org.datanucleus.samples.store.Product prod");
```

This is known as an explicit variable, since it is declared explicitly. Multiple variables can be declared using semi-colon (;) to separate variable declarations.

```
query.declareVariables("String var1; String var2");
```

JDO2 also defines implicit variables, where we skip the declareVariables call and rely on the JDO implementation to work out the type of the variable.

setCandidates()

Set the candidate Collection to query.

This is applicable to JDOQL.

setOrdering()

Set the ordering specification for the result Collection. The ordering specification is a String containing one or more ordering declarations separated by commas. Each ordering declaration is the name of the field on which to order the results followed by one of the following words: "ascending" or "descending". The field must be declared in the candidate class or must be a navigation expression starting with a field in the candidate class.

This is applicable to JDOQL.

With JDOQL you can specify the ordering using the normal JDOQL syntax for a parameter, and then add ascending or descending (UPPER or lower case are both valid) are to give the direction. In addition the abbreviated forms of asc and desc (again, UPPER and lower case forms are accepted) to save typing. For example, you may set the ordering as follows

```
query.setOrdering("productId DESC");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

setGrouping()

Set the grouping expressions, optionally including a "having" clause. When grouping is specified, each result expression must either be an expression contained in the grouping, or an aggregate evaluated once per group.

This is applicable to JDOQL.

addExtension(), setExtensions()

JDO's query API allows implementations to support extensions and provides a simple interface for enabling the use of such extensions on queries.

```
q.addExtension("extension_name", "value");
```

```
    HashMap exts = new HashMap();
    exts.put("extension1", value1);
    exts.put("extension2", value2);
    q.setExtensions(exts);
```

getFetchPlan()

When a Query is executed it executes in the datastore, which returns a set of results. DataNucleus could clearly read all results from this ResultSet in one go and return them all to the user, or could allow control over this fetching process. JDO2 provides a fetch size on the Fetch Plan to allow this control. You would set this as follows

```
Query q = pm.newQuery(...);
q.getFetchPlan().setFetchSize(FetchPlan.FETCH_SIZE_OPTIMAL);
```

fetch size has 3 possible values.

- FETCH_SIZE_OPTIMAL - allows DataNucleus full control over the fetching. In this case DataNucleus will fetch each object when they are requested, and then when the owning transaction is committed will retrieve all remaining rows (so that the Query is still usable after the close of the transaction).
- FETCH_SIZE_GREEDY - DataNucleus will read all objects in at query execution. This can be efficient for queries with few results, and very inefficient for queries returning large result sets.
- A positive value - DataNucleus will read this number of objects at query execution. Thereafter it will read the objects when requested.

In addition to the number of objects fetched, you can also control which fields are fetched for each object of the candidate type. This is controlled via the FetchPlan. See also Fetch Groups.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DataNucleus also allows an extension to give further control. As mentioned above, when the transaction containing the Query is committed, all remaining results are read so that they can then be accessed later (meaning that the query is still usable). Where you have a large result set and you don't want this behaviour you can turn it off by specifying a Query extension

```
q.addExtension("datanucleus.query.loadResultsAtCommit", "false");
```

so when the transaction is committed, no more results will be available from the query.

In some situations you don't want all FetchPlan fields retrieving, and DataNucleus provides an extension to turn this off, like this

```
q.addExtension("datanucleus.query.useFetchPlan", "false");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.2 Named Queries

JDO Named Queries

With the JDO2 API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called Product (something to sell in a store). We define the JDO Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```
<jdo>
    <package name="org.datanucleus.example">
        <class name="Product">
            ...
            <query name="SoldOut" language="javax.jdo.query.JDOQL"><![CDATA[
            SELECT FROM org.datanucleus.example.Product WHERE status == "Sold Out"
            ]]></query>
        </class>
    </package>
</jdo>
```

So we have a JDOQL query called "SoldOut" defined for the class Product that returns all Products (and subclasses) that have a status of "Sold Out". Out of interest, what we would then do in our application to execute this query woule be

```
Query query=pm.newNamedQuery(org.datanucleus.example.Product.class,"SoldOut");
Collection results=(Collection)query.execute();
```

The above example was for the JDOQL object-based query language. We can do a similar thing using SQL, so we define the following in our MetaData for our Product class

```
<jdo>
    <package name="org.datanucleus.example">
        <class name="Product">
            ...
            <query name="PriceBelowValue" language="javax.jdo.query.SQL"><![CDATA[
            SELECT NAME FROM PRODUCT WHERE PRICE < ?
            ]]></query>
        </class>
    </package>
</jdo>
```

So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

query, asking for the names of all Products with price below 20 euros.

```
Query
query=pm.newNamedQuery(org.datanucleus.example.Product.class,"PriceBelowValue");
Collection results=(Collection)query.execute(20.0);
```

All of the examples above have been specifed within the <class> element of the MetaData. You can, however, specify queries below <jdo> in which case the query is not scoped by a particular candidate class. In this case you must put your queries in any of the following MetaData files

```
/META-INF/package.jdo
/WEB-INF/package.jdo
/package.jdo
/META-INF/package-{mapping}.orm
/WEB-INF/package-{mapping}.orm
/package-{mapping}.orm
/META-INF/package.jdoquery
/WEB-INF/package.jdoquery
/package.jdoquery
```

Please proceed to the sections specific to JDOQL and SQL for details on the precise nature of the query for the querying languages supported by DataNucleus.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.3 JDOQL

JDOQL Queries

JDO defines ways of querying objects persisted into the datastore. It provides its own object-based query language (JDOQL). JDOQL is designed as the Java developers way of having the power of SQL queries, yet retaining the Java object relationship that exist in their application model. DataNucleus provides all functionality required by the JDO 1.0 and 2.0 specifications as well as providing a series of vendor extensions. A typical JDOQL query may be set up in one of 2 ways. Here's an example

```
Declarative JDOQL :
Query q = pm.newQuery(org.datanucleus.Person.class, "lastName == \"Jones\" && age <
age_limit");
q.declareParameters("double age_limit");
List results = (List)q.execute(20.0);

Single-String JDOQL :
Query q = pm.newQuery("SELECT FROM org.datanucleus.Person WHERE lastName ==
\"Jones\"" +
                      " && age < :age_limit PARAMETERS double age_limit");
List results = (List)q.execute(20.0);
```

So here in our example we select all "Person" objects with surname of "Jones" and where the persons age is below 20. The language is intuitive for Java developers, and is intended as their interface to accessing the persisted data model. As can be seen above, the query is made up of distinct parts. The class being selected (the SELECT clause in SQL), the filter (which equates to the WHERE clause in SQL), together with any sorting (the ORDER BY clause in SQL), etc.

Single-String JDOQL

In traditional (declarative) JDOQL (JDO 1.0) it was necessary to specify the component parts (filter, candidate class, ordering, etc) of the query using the mutator methods on the Query. In JDO 2 you can now specify it all in a single string. This string has to follow a particular pattern, but provides the convenience that many people have been asking for. The pattern to use is as follows

```
SELECT [UNIQUE] [<result>] [INTO <result-class>]
        [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
        [WHERE <filter>]
        [VARIABLES <variable declarations>]
        [PARAMETERS <parameter declarations>]
        [<import declarations>]
        [GROUP BY <grouping>]
        [ORDER BY <ordering>]
        [RANGE <start>, <end>]
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The "keywords" in the query are shown in UPPER CASE but can be in UPPER or lower case.

Lets give an example of a query using this syntax

```
SELECT UNIQUE FROM org.datanucleus.samples.Employee ORDER BY departmentNumber
```

so we form the parts of the query as before, yet here we just specify it all in a single call.

Accessing Fields

In JDOQL you access fields in the query by referring to the field name. For example, if you are querying a class called Product and it has a field "price", then you access it like this

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class, "price <
150.0");
```

In addition to the persistent fields, you can also access "public static final" fields of any class. You can do this as follows

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,
       "taxPercent < org.datanucleus.samples.store.Product.TAX_BAND_A");
```

So this will find all products that include a tax percentage less than some "BAND A" level. Where you are using "public static final" fields you can either fully-qualify the class name or you can include it in the "imports" section of the query (see later).

Data types : literals

JDOQL supports the following literals: IntegerLiteral, FloatingPointLiteral, BooleanLiteral, CharacterLiteral, StringLiteral, and NullLiteral.

Operators precedence

The following list describes the operator precedence in JDOQL.

1. Cast
2. Unary ("~") ("!")
3. Unary ("+") ("-")
4. Multiplicative ("*") ("/") ("%")
5. Additive ("+") ("-")
6. Relational (">=") (">") ("<=") ("<") ("instanceof")

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

7. Equality ("==") ("=!")
8. Boolean logical AND ("&")
9. Boolean logical OR ("|")
10. Conditional AND ("&&")
11. Conditional OR ("||")

Concatenation Expressions

The concatenation operator(+) concatenates a String to either another String or Number. Concatenations of String or Numbers to null results in null.

Example 1 - Use of Explicit Parameters

Here's a simple example for finding the elements of a class with a field below a particular threshold level. Here we pass in the threshold value (limit), and sort the output in order of ascending price.

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,"price <
limit");
query.declareParameters("double limit");
query.setOrdering("price ascending");
List results = (List)query.execute(150.00);

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE "
+
                "price < limit PARAMETERS double limit ORDER BY price ASCENDING");
List results = (List)query.execute(150.00);
```

For completeness, the class is shown here

```
class Product
{
    String name;
    double price;
    java.util.Date endDate;
    ...
}

<jdo>
    <package name="org.datanucleus.samples.store">
        <class name="Product">
            <field name="name">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="abreviation">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
            <field name="price"/>
            <field name="endDate"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        </class>
    </package>
</jdo>
```

Example 2 - Use of Implicit Parameters

Let's repeat the previous query but this time using *implicit* parameters.

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,"price <
:limit");
query.setOrdering("price ascending");
List results = (List)query.execute(150.00);

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE "
+
                "price < :limit ORDER BY price ASCENDING");
List results = (List)query.execute(150.00);
```

So we omitted the declaration of the parameter and just prefixed it with a colon (:)

Example 3 - Comparison against Dates

Here's another example using the same Product class as above, but this time comparing to a Date field. Because we are using a type in our query, we need to import it ... just like you would in a Java class if you were using it there.

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,
                          "endDate > best_before_limit");
query.declareImports("import java.util.Date");
query.declareParameters("Date best_before_limit");
query.setOrdering("endDate descending");
Collection results = (Collection)query.execute(my_date_limit);

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product " +
                "WHERE endDate > best_before_limit " +
                "PARAMETERS Date best_before_limit " +
                "import java.util.Date ORDER BY endDate DESC");
List results = (List)query.execute(my_date_limit);
```

Example 4 - Instanceof

This example demonstrates use of the "instanceof" operator. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. Clearly the field "b" of A can be of type B, B1, B2, B3 etc, and we want to find all objects of type A that have the field "b" that is of type B2. We do it like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.A.class);
query.setFilter("b instanceof org.datanucleus.samples.B2");
List results = (List)query.execute();

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.A WHERE b instanceof
org.datanucleus.samples.B2");
List results = (List)query.execute();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.4 JDOQL : Result

JDOQL : Result

As we have seen, a JDOQL query is made up of different parts. In this section we look at the result part of the query. The result is what we want returning. By default (when not specifying the result) the objects returned will be of the candidate class type, where they match the query filter. Firstly let's look at what you can include in the result clause.

- this - the candidate instance
- A field name
- A variable
- A parameter (though why you would want a parameter returning is hard to see since you input the value in the first place)
- An aggregate (count(), avg(), sum(), min(), max())
- An expression involving a field (e.g "field1 + 1")
- A navigational expression (navigating from one field to another ... e.g "field1.field4")

The result is specified in JDOQL like this

```
query.setResult("count(field1), field2");
```

In Single-String JDOQL you would specify it directly.

Result type

What you specify in the result defines what form of result you get back.

- Object - this is returned if you have only a single row in the results and a single column. This is achived when you specified either UNIQUE, or just an aggregate (e.g "max(field2)")
- Object[] - this is returned if you have only a single row in the results, but more than 1 column (e.g "max(field1), avg(field2)")
- List<Object> - this is returned if you have only a single column in the result, and you don't have only aggregates in the result (e.g "field2")
- List<Object[]> - this is returned if you have more than 1 column in the result, and you don't have only aggregates in the result (e.g "field2, avg(field3)")

Aggregates

There are situations when you want to return a single number for a column, representing an aggregate of the values of all records. There are 5 standard JDO2 aggregate functions available. These are

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- avg(val) - returns the average of "val". "val" can be a field, numeric field expression or "distinct field".
- sum(val) - returns the sum of "val". "val" can be a field, numeric field expression, or "distinct field".
- count(val) - returns the count of records of "val". "val" can be a field, or can be "this", or "distinct field".
- min(val) - returns the minimum of "val". "val" can be a field
- max(val) - returns the maximum of "val". "val" can be a field

So to utilise these you could specify something like

```
Query q = pm.newQuery("SELECT max(price), min(price) FROM
org.datanucleus.samples.store.Product WHERE status == 1");
```

This will return a single row of results with 2 values, the maximum price and the minimum price of all products that have status code of 1.

Result MetaData

DataNucleus provides a useful extension if you wish to know the types of the result from a JDOQL (or JPQL) query. You do as follows

```
Query q = pm.newQuery(...);
q.compile();
QueryResultsMetaData qmd = NucleusJDOHelper.getMetaDataForQueryResults(q);
int num = qmd.getExpressionCount();
for (int i=0;i<num;i++)
{
    Class type = qmd.getExpressionType(i);
}
```

so we now have the number of expressions returned, and the type of each one.

Example 1 - Use of aggregates

JDO 2 introduces the ability to use aggregates in queries. Here's another example using the same Product class as above, but this time looking for the maximum price of products that are CD Players. Note that the result for this particular query will be of type Double since there is a single double precision value being returned via the "result".

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setFilter("name == \"CD Player\"");
query.setResult("max(this.price)");
List results = (List)query.execute();
Iterator iter = c.iterator();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Double max_price = (Double)iter.next();

Single-String JDOQL :
Query query = pm.newQuery("SELECT max(price) FROM
org.datanucleus.samples.store.Product WHERE name == \"CD Player\"");
List results = (List)query.execute();
Iterator iter = c.iterator();
Double max_price = (Double)iter.next();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.5 JDOQL : Methods

JDOQL : Methods

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. This document defines the standard methods available in JDO2. If you look at the datastore-specific implementations you can find some extensions to this list (particularly for RDBMS).

Java Type	Method	Description	Specification
String	startsWith(String)	Returns if the string starts with the passed string	JDO 1.0, JDO 2.0
String	endsWith(String)	Returns if the string ends with the passed string	JDO 1.0, JDO 2.0
String	indexOf(String)	Returns the first position of the passed string	JDO 2.0
String	indexOf(String,int)	Returns the position of the passed string, after the passed position	JDO 2.0
String	substring(int)	Returns the substring starting from the passed position	JDO 2.0
String	substring(int,int)	Returns the substring between the passed positions	JDO 2.0
String	toLowerCase()	Returns the string in lowercase	JDO 2.0
String	toUpperCase()	Retuns the string in UPPERCASE	JDO 2.0
String	matches(String pattern)	Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method.	JDO 2.0
Collection	isEmpty()	Returns whether the collection is empty	JDO 1.0, JDO 2.0
Collection	contains(value)	Returns whether the collection contains the passed element	JDO 1.0, JDO 2.0
Collection	size()	Returns the number of elements in the collection	JDO 2.0
Map	isEmpty()	Returns whether the map is empty	JDO 1.0, JDO 2.0
Map	containsKey(key)	Returns whether the map contains the passed key	JDO 2.0
Map	containsValue(value)	Returns whether the map contains the passed value	JDO 2.0

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification
Map	get(key)	Returns the value from the map with the passed key	JDO 2.0
Map	size()	Returns the number of entries in the map	JDO 2.0
Math	abs(number)	Returns the absolute value of the passed number	JDO 2.0
Math	sqrt(number)	Returns the square root of the passed number	JDO 2.0
JDOHelper	getObjectId(object)	Returns the object identity of the passed persistent object	JDO 2.0

The following sections provide some examples of what can be done using JDOQL methods.

Example 1 - Map methods (I)

Here's another example using the same Product class as a value in a Map. This introduces how you query Collection and Map fields using the operations available. Collections and Maps act very similarly. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with the name "product 1".

```
Declarative JDOQL :
Extent e=pm.getExtent(org.datanucleus.samples.store.Inventory.class,false);
Query query = pm.newQuery(e,"products.containsKey(\"product 1\")");
List results = (List)query.execute();

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Inventory
EXCLUDE SUBCLASSES " +
                "WHERE products.containsKey(\"product 1\")");
List results = (List)query.execute();
```

Here's the source code for reference

```
class Inventory
{
    Map products;
    ...
}
class Product
{
    String name;
    double price;
    double salePrice;
    java.util.Date endDate;
    String[] composition;
    ...
}
```

```
<jdo>
    <package name="org.datanucleus.samples.store">
        <class name="Inventory">
            <field name="products">
                <map key-type="java.lang.String"
value-type="org.datanucleus.samples.store.Product"/>
                <key mapped-by="name"/>
            </field>
        </class>

        <class name="Product">
            <field name="name">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="price"/>
            <field name="endDate"/>
        </class>
    </package>
</jdo>
```

Example 2 - Map methods (II)

We might want to to check if a Collection field contains one or other elements. We extend the previous example that is using the Product class as a value in a Map. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with the name "product 1" or "product 2".

```
Declarative JDOQL :
Extent e=pm.getExtent(org.datanucleus.samples.store.Inventory.class,false);
Query query = pm.newQuery(e);
query.declareVariables("String productName");
query.setFilter("products.containsKey(productName) && (productName==\"product 1\" ||
productName==\"product 2\")");
List results = (List)query.execute();

Single-String JDOQL:
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Inventory
EXCLUDE SUBCLASSES " +
    "WHERE products.containsKey(productName) && (productName==\"product 1\" ||
productName==\"product 2\") " +
    "VARIABLES String productName");
List results = (List)query.execute();
```

Example 3 - String.startsWith() method

Here's another example using the same Product class as above, but this time looking for objects which their abreviation is the begin of a trade name. The trade name is provided as parameter.

```
Declarative JDOQL :
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.declareImports("import java.lang.String");
query.declareParameters("java.lang.String tradeName");
query.setFilter("tradeName.startsWith(this.abbreviation)");
List results = (List)query.execute("Workbook Advanced");

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product " +
                "WHERE tradeName.startsWith(this.abbreviation) " +
                "PARAMETERS java.lang.String tradeName import java.lang.String");
List results = (List)query.execute("Workbook Advanced");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.6 JDOQL : Subqueries

JDOQL Subqueries

With JDOQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JDO 2.1 provides a very useful addition to JDOQL adding on subqueries, so that both calculations can be performed in one query. Here's an example, using single-string JDOQL

```
SELECT FROM org.datanucleus.Employee WHERE salary >
    (SELECT avg(salary) FROM org.datanucleus.Employee e)
```

So we want to find all Employees that have a salary greater than the average salary. In single-string JDOQL the subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "e", whereas in the outer query the alias is "this".

We can specify the same query using the JDOQL API, like this

```
Query averageSalaryQuery = pm.newQuery(Employee.class);
averageSalaryQuery.setResult("avg(this.salary)");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");
q.declareVariables("double averageSalary");
q.addSubquery(averageSalaryQuery, "double averageSalary", null, null);
List results = (List)q.execute();
```

So we define a subquery as its own Query (that could be executed just like any query if so desired), and the in the main query have an implicit variable that we define as being represented by the subquery.

Referring to the outer query in the subquery

JDOQL subqueries allows use of the outer query fields within the subquery if so desired. Taking the above example and extending it, here is how we do it in single-string JDOQL

```
SELECT FROM org.datanucleus.Employee WHERE salary >
    (SELECT avg(salary) FROM org.datanucleus.Employee e WHERE e.lastName ==
this.lastName)
```

So with single-string JDOQL we make use of the alias identifier "this" to link back to the outer query.

Using the JDOQL API, to achieve the same thing we would do

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Query averageSalaryQuery = pm.newQuery(Employee.class);
averageSalaryQuery.setResult("avg(this.salary)");
averageSalaryQuery.setFilter("this.lastName == :lastNameParam");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");
q.declareVariables("double averageSalary");
q.addSubquery(averageSalaryQuery, "double averageSalary", null, "this.lastName");
List results = (List)q.execute();
```

So with the JDOQL API we make use of parameters, and the last argument to *addSubquery* is the value of the parameter *lastNameParam*.

Candidate of the subquery being part of the outer query

There are occasions where we want the candidate of the subquery to be part of the outer query, so JDOQL subqueries has the notion of a *candidate expression*. This is an expression relative to the candidate of the outer query. An example

```
SELECT FROM org.datanucleus.Employee WHERE this.weeklyhours >
    (SELECT AVG(e.weeklyhours) FROM this.department.employees e)
```

so the candidate of the subquery is *this.department.employees*. If using a candidate expression we must provide an alias.

You can do the same with the JDOQL API. Like this

```
Query averageHoursQuery = pm.newQuery(Employee.class);
averageHoursQuery.setResult("avg(this.weeklyhours)");

Query q = pm.newQuery(Employee.class);
q.setFilter("this.weeklyhours > averageWeeklyhours");
q.addSubquery(averageHoursQuery, "double averageWeeklyhours",
"this.department.employees", null);
```

so now our subquery has a candidate related to the outer query candidate.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.7 JDOQL : In-Memory

JDOQL : In-Memory queries

The typical use of a JDOQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JDOQL syntax in the datastore *native query language* and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of *in-memory evaluation* in DataNucleus.

- Query methods **Collection.contains**, **Map.containsKey**, **Map.containsValue** and **Map.get** are not supported currently
- Only simple subqueries are supported currently

To enable evaluation in memory you specify the query extension **datanucleus.query.evaluateInMemory** to *true* as follows

```
query.addExtension("datanucleus.query.evaluateInMemory","true");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.8 SQL

SQL Queries

The ability to query the datastore is an essential part of any system that persists data. Sometimes an object-based query language (such as JDOQL) is considered not suitable, maybe due to the lack of familiarity of the application developer with such a query language. In this case it is desirable to query using SQL. JDO 2 standardises this as a valid query mechanism, and DataNucleus supports this. **Please be aware that the SQL query that you invoke has to be valid for your RDBMS, and that the SQL syntax differs across almost all RDBMS.**

To utilise SQL syntax in queries, you create a Query as follows

```
Query q = pm.newQuery("javax.jdo.query.SQL",the_query);
```

You have several forms of SQL queries, depending on what form of output you require.

- **No candidate class and no result class** - the result will be a List of Objects (when there is a single column in the query), or a List of Object[]s (when there are multiple columns in the query)
- **Candidate class specified, no result class** - the result will be a List of candidate class objects, or will be a single candidate class object (when you have specified "unique"). The columns of the querys result set are matched up to the fields of the candidate class by name. You need to select a minimum of the PK columns in the SQL statement.
- **No candidate class, result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). Your result class has to abide by the rules of JDO2 result classes (see Result Class specification) - this typically means either providing public fields matching the columns of the result, or providing setters/getters for the columns of the result.
- **Candidate class and result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). The result class has to abide by the rules of JDO2 result classes (see Result Class specification).

Stored Procedures

In JDO2 all SQL queries must begin "SELECT ...", and consequently it is not possible to execute stored procedures. In DataNucleus we have an extension that allows this to be overridden. To enable this you should pass the property datanucleus.rdbms.sql.allowAllSQLStaments as true when creating the PersistenceManagerFactory. Thereafter you just invoke your stored procedures like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "EXECUTE sp_who");
```

Where "sp_who" is the stored procedure being invoked. Clearly the same rules will apply regarding the results of the stored procedure and mapping them to any result class. The syntax of calling a stored

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

procedure differs across RDBMS. Some require "CALL ..." and some "EXECUTE ...". Go consult your manual.

Inserting/Updating/Deleting

In JDO2 all SQL queries must begin "SELECT ...", and consequently it is not possible to execute queries that change data. In DataNucleus we have an extension that allows this to be overridden. To enable this you should pass the property datanucleus.rdbms.sql.allowAllSQLStaments as true when creating the PersistenceManagerFactory. Thereafter you just invoke your statements like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "UPDATE MY_TABLE SET MY_COLUMN = ?
WHERE MY_ID = ?");
```

you then pass any parameters in as normal for an SQL query.

Parameters

In JDO2 SQL queries can have parameters but must be *positional*. This means that you do as follows

```
Query q = pm.newQuery("javax.jdo.query.SQL",
    "SELECT col1, col2 FROM MYTABLE WHERE col3 = ? AND col4 = ? and col5 = ?");
List results = (List) q.execute(val1, val2, val3);
```

So we used traditional JDBC form of parametrisation, using "?".

DataNucleus also supports two further variations. The first is called *numbered* parameters where we assign numbers to them, so the previous example could have been written like this

```
Query q = pm.newQuery("javax.jdo.query.SQL",
    "SELECT col1, col2 FROM MYTABLE WHERE col3 = ?1 AND col4 = ?2 and col5 = ?1");
List results = (List) q.execute(val1, val2);
```

so we can reuse parameters in this variation. The second variation is called *named* parameters where we assign names to them, and so the example can be furtehr rewritten like this

```
Query q = pm.newQuery("javax.jdo.query.SQL",
    "SELECT col1, col2 FROM MYTABLE WHERE col3 = :firstVal AND col4 = :secondVal and
col5 = :firstVal");
Map params = new HashMap();
params.put("firstVal", val1);
params.put("secondVal", val1);
List results = (List) q.executeWithMap(params);
```

Example 1 - Using SQL aggregate functions, without candidate class

Here's an example for getting the size of a table without a candidate class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM MYTABLE");
List results = (List) query.execute();
Integer tableSize = (Integer) result.iterator().next();
```

Here's an example for getting the maximum and miminum of a parameter without a candidate class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT max(PARAM1), min(PARAM1)
FROM MYTABLE");
List results = (List) query.execute();
Object[] measures = (Object[])result.iterator().next();
Double maximum = (Double)measures[0];
Double minimum = (Double)measures[1];
```

Example 2 - Using SQL aggregate functions, with result class

Here's an example for getting the size of a table with a result class. So we have a result class of

```
public class TableStatistics
{
    private int total;

    public setTotal(int total);
}
```

So we define our query to populate this class

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) AS total FROM
MYTABLE");
query.setResultClass(TableStatistics.class);
List results = (List) query.execute();
TableStatistics tableStats = (TableStatistics) result.iterator().next();
```

Each row of the results is of the type of our result class. Since our query is for an aggregate, there is actually only 1 row.

Example 3 - Retrieval using candidate class

When we want to retrieve objects of a particular PersistenceCapable class we specify the candidate class. Here we need to select, as a minimum, the identity columns for the class.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Query query = pm.newQuery("javax.jdo.query.SQL",
                        "SELECT MY_ID, MY_NAME FROM MYTABLE");
query.setClass(MyClass.class);
List results = (List) query.execute();
Iterator resultsIter = results.iterator();
while (resultsIter.hasNext())
{
    MyClass obj = (MyClass)resultsIter.next();
}
```

```
class MyClass
{
    String name;
    ...
}

<jdo>
    <package name="org.datanucleus.samples.sql">
        <class name="MyClass" identity-type="datastore" table="MYTABLE">
            <datastore-identity strategy="identity">
                <column name="MY_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="MY_NAME"/>
            </field>
        </class>
    </package>
</jdo>
```

Example 4 - Using parameters, without candidate class

Here's an example for getting the number of people with a particular email address. You simply add a "?" for all parameters that are passed in, and these are subsitituted at execution time.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM PERSON WHERE
EMAIL_ADDRESS = ?");
List results = (List) query.execute("nobody@datanucleus.org");
Integer tableSize = (Integer) result.iterator().next();
```

Example 5 - Named Query

While "named" queries were introduced primarily for JDOQL queries, we can define "named" queries for SQL also. So let's take a Product class, and we want to define a query for all products that are "sold out". We firstly add this to our MetaData

```
<jdo>
    <package name="org.datanucleus.samples.store">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <class name="Product" identity-type="datastore" table="PRODUCT">
            <datastore-identity strategy="identity">
                <column name="PRODUCT_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="NAME"/>
            </field>
            <field name="status" persistence-modifier="persistent">
                <column name="STATUS"/>
            </field>

            <query name="SoldOut" language="javax.jdo.query.SQL"><![CDATA[
                SELECT PRODUCT_ID FROM PRODUCT WHERE STATUS == "Sold Out"
            ]]></query>
        </class>
    </package>
</jdo>
```

And then in our application code we utilise the query

```
Query q = pm.newNamedQuery(Product.class, "SoldOut");
List results = (List)q.execute();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

10.9 JPQL

JPQL Queries

JDO provides a flexible API for use of query languages. DataNucleus makes use of this to allow use of the query language defined in the JPA1 specification (JPQL) with JDO persistence. To provide a simple example, this is what you would do

```
Query q = pm.newQuery("javax.jdo.query.JPQL", "SELECT p FROM Person p WHERE
p.lastName = 'Jones'");
List results = (List)q.execute();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

JPQL Syntax

In traditional (declarative) JDOQL (JDO 1.0) it was necessary to specify the component parts (filter, candidate class, ordering, etc) of the query using the mutator methods on the Query. In JDO 2 you can now specify it all in a single string. This string has to follow a particular pattern, but provides the convenience that many people have been asking for. The pattern to use is as follows

```
SELECT [<result>]
    [FROM <candidate-class(es)>]
    [WHERE <filter>]
    [GROUP BY <grouping>]
    [HAVING <having>]
    [ORDER BY <ordering>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

Entity Name

In the example shown you note that we did not specify the full class name. We used *Person p* and thereafter could refer to *p* as the alias. The *Person* is called the **entity name** and in JPA MetaData this can be defined against each class in its definition. With JDO we dont have this MetaData attribute so we simply define the **entity name** as *the name of the class omitting the package name*. So *org.datanucleus.test.samples.Person* will have an entity name of *Person*.

Input Parameters

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

different values. Let's take two examples

```
Named Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :surname AND
o.firstName = :forename");
q.setParameter("surname", theSurname);
q.setParameter("forename", theForename");

Numbered Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND
p.firstName = ?2");
q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

So in the first case we have parameters that are prefixed by **:** (colon) to identify them as a parameter and we use that name when calling *Query.setParameter()*. In the second case we have parameters that are prefixed by **?** (question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*.

Unique Results

Sometimes you know that the query can only every return 0 or 1 objects. In this case you can simplify your job by adding

```
query.setUnique(true);
```

In this case the return from the execution of the Query will be a single Object rather than a List, so you've no need to use iterators, just cast it to your candidate class type.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.1 JPA API

JPA API

The work in this section can be split into several sections.

- With JPA you need to create an EntityManagerFactory to connect to a datastore
- With JPA you need to create an EntityManager to provide the interface to persisting/accessing objects
- Controlling the transaction
- Accessing persisted object via queries, using JPQL, or SQL

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.2 Entity Manager Factory

Entity Manager Factory

Any JPA-enabled application will require at least one EntityManagerFactory. Typically applications create one per datastore being utilised. An EntityManagerFactory provides access to EntityManagers which allow objects to be persisted, and retrieved. The EntityManagerFactory can be configured to provide particular behaviour.

The simplest way of creating an EntityManagerFactory in a J2SE environment is as follows

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

...

EntityManagerFactory emf =
Persistence.createEntityManagerFactory(persistenceUnitName);
```

So you simply provide the name of the persistence-unit which defines the properties, classes, metadata etc to be used. An alternative is to specify the properties to use along with the *persistence-unit* name. In that case the passed properties will override any that are specified for the persistence unit itself.

Standard JPA Properties

Parameter	Values	Description
javax.persistence.provider		Class name of the provider to use. DataNucleus has a provider name of org.datanucleus.jpa.PersistenceProviderImpl
javax.persistence.transactionType	RESOURCE_LOCAL | JTA	Type of transactions to use. In J2SE the default is RESOURCE_LOCAL. In J2EE the default is JTA.
javax.persistence.jtaDataSource		JNDI name of a (transactional) JTA data source.
javax.persistence.nonJtaDataSource		JNDI name of a (non-transactional) data source.
javax.persistence.jdbc.driver		Alias for datanucleus.ConnectionDriverName
javax.persistence.jdbc.url		Alias for datanucleus.ConnectionURL
javax.persistence.jdbc.user		Alias for datanucleus.ConnectionUserName
javax.persistence.jdbc.password		Alias for datanucleus.ConnectionPassword
javax.persistence.query.timeout		Alias for datanucleus.query.timeout

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Extension DataNucleus Properties

DataNucleus provides many properties to extend the control that JPA gives you. These can be used alongside the above standard JPA properties, but will only work with DataNucleus. Please consult the Persistence Properties Guide for full details. In particular, look at the properties for specifying the datastore to be used including **datanucleus.ConnectionURL**, **datanucleus.ConnectionDriver**, **datanucleus.ConnectionUserName**, **datanucleus.ConnectionPassword**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.3 Entity Manager

Entity Manager

As you read in the guide for EntityManagerFactory, to control the persistence of your objects you will require at least one EntityManagerFactory. Once you have obtained this object you then use this to obtain an EntityManager. An EntityManager provides access to the operations for persistence of your objects. This short guide will demonstrate some of the more common operations.

You obtain an EntityManager as follows

```
EntityManager em = emf.createEntityManager();
```

In general you will be performing all operations on a EntityManager within a transaction, whether your transactions are controlled by your J2EE container, by a framework such as Spring, or by locally defined transactions. In the examples below we will omit the transaction demarcation for clarity.

Persisting an Object

The main thing that you will want to do with the data layer of a JPA-enabled application is persist your objects into the datastore. As we mentioned earlier, a EntityManagerFactory represents the datastore where the objects will be persisted. So you create a normal Java object in your application, and you then persist this as follows

```
em.persist(obj);
```

This will result in the object being persisted into the datastore, though clearly it will not be persistent until you commit the transaction. The LifecycleState of the object changes from Transient to PersistentClean (after persist()), to Hollow (at commit).

Using an Objects identity

Once you have persisted an object, it has an "identity". This is a unique way of identifying it. When you specify the persistence for the class you specified an id class so you can create the identity from that. So what ? Well the identity can be used to retrieve the object again at some other part in your application. So you pass the identity into your application, and the user clicks on some button on a web page and that button corresponds to a particular object identity. You can then go back to your data layer and retrieve the object as follows

```
Object obj = em.find(cls, id);
```

where *cls* is the class of the object you want to find, and *id* is the identity.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Deleting an Object

When you need to delete an object that you had previous persisted, deleting it is simple. Firstly you need to get the object itself, and then delete it as follows

```
Object obj = em.find(cls, id);  // Retrieves the object to delete
em.remove(obj);
```

Modifying a persisted Object

To modify a previously persisted object you take the object and update it in your code. When you are ready to persist the changes you do the following

```
Object updatedObj = em.merge(obj)
```

Refreshing a persisted Object

When you think that the datastore has more up-to-date values than the current values in a retrieved persisted object you can refresh the values in the object by doing the following

```
em.refresh(obj)
```

This will do the following

- Refresh all fields that are to be eagerly fetched from the datastore
- Unload all loaded fields that are to be lazily fetched.

If the object had any changes they will be thrown away by this step, and replaced by the latest datastore values.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.4 Datastore Control

Datastore Control

Sometimes a datastore has particular requirements, and maybe is out of direct control of an application being managed by a different group. In this situation you may have to configure the datastore layer to meet these requirements.

Read-Only

If your datastore is read-only (you can't add/update/delete any data in it), obviously you could just configure your application to not perform these operations. An alternative is to set the EntityManagerFactory as "read-only". You do this by setting the persistence property **datanucleus.readOnlyDatastore** to *true*.

From now on, whenever you perform a persistence operation that implies a change in datastore data, the JPA operation will throw an exception. You can change this behaviour using the persistence property *datanucleus.readOnlyDatastoreAction* with values of "EXCEPTION" (default), and "IGNORE". "IGNORE" has the effect of simply ignoring all attempted updates to readonly objects.

Fixed Schema

Some datastores have a "schema" defining the structure of data stored within that datastore. In this case we don't want to allow updates to the structure at all. You can set this when creating your EntityManagerFactory by setting the persistence property **datanucleus.fixedDatastore** to *true*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.5 Datastore Replication

Datastore Replication

Many applications make use of multiple datastores. It is a common requirement to be able to replicate parts of one datastore in another datastore. Obviously, depending on the datastore, you could make use of the datastores own capabilities for replication. DataNucleus provides its own extension to JPA to allow replication from one datastore to another. This extension doesn't restrict you to using 2 datastores of the same type. You could replicate from RDBMS to XML for example, or from MySQL to HSQLDB.

The following sample code will replicate all objects of type *Product* and *Employee* from EMF1 to EMF2. These EMFs are created in the normal way so, as mentioned above, EMF1 could be for a MySQL datastore, and EMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```
import org.datanucleus.jpa.JPAReplicationManager;

...

JPAReplicationManager replicator = new JPAReplicationManager(emf1, emf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.6 Transaction Types

Transaction Types

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. There are the following types of transaction.

- Transactions can lock all records in a datastore and keep them locked until they are ready to commit their changes. These are known as Pessimistic (or datastore) Transactions.
- Transactions can simply assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. These are known as Optimistic Transactions.
- Non-transactional where you perform operations effectively in "auto-commit" mode

Pessimistic (Datastore) Transactions

Pessimistic transactions arent supported in JPA but are provided as a vendor extension. They are suitable for short lived operations where no user interaction is taking place and so it is possible to block access to datastore entities for the duration of the transaction. You would select pessimistic transactions by adding the persistence property **datanucleus.Optimistic** as *false*.

By default DataNucleus does not currently lock the objects fetched in a pessimistic transaction, but you can configure this behaviour for RDBMS datastores by setting the persistence property datanucleus.rdbms.useUpdateLock to true. This will result in all "SELECT ... FROM ..." statements being changed to be "SELECT ... FROM ... FOR UPDATE". This will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax.

With a pessimistic transaction DataNucleus will grab a datastore connection at the first operation, and maintain it for the duration of the transaction. A single connection is used for the transaction (with the exception of any Identity Generation operations which need datastore access, so these can use their own connection).

In terms of the process of a pessimistic (datastore) transaction, we demonstrate this below.

Operation	DataNucleus process	Datastore process
Start transaction		
Persist object	Prepare object (1) for persistence	Open connection. Insert the object (1) into the datastore
Update object	Prepare object (2) for update	Update the object (2) into the datastore
Persist object	Prepare object (3) for persistence	Insert the object (3) into the datastore
Update object	Prepare object (4) for update	Update the object (4) into the datastore

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Operation	DataNucleus process	Datastore process
Flush	No outstanding changes so do nothing	
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	Insert the object (5) into the datastore
Update object	Prepare object (6) for update	Update the object (6) into the datastore
Commit transaction		Commit connection

So here whenever an operation is performed, DataNucleus pushes it straight to the datastore. Consequently any queries will always reflect the current state of all objects in use. However this mode of operation has no version checking of objects and so if they were updated by external processes in the meantime then they will overwrite those changes.

It should be noted that DataNucleus provides two persistence properties that allow an amount of control over when flushing happens with datastore transactions.

- *datanucleus.datastoreTransactionDelayOperations* when set to true will try to delay all datastore operations until commit/flush.
- *datanucleus.datastoreTransactionFlushLimit* represents the number of dirty objects before a flush is performed. This defaults to 1.

Optimistic Transactions

Optimistic transactions are the only official option in JPA. They are suitable for longer lived operations maybe where user interaction is taking place and where it would be undesirable to block access to datastore entities for the duration of the transaction. The assumption is that data altered in this transaction will not be updated by other transactions during the duration of this transaction, so the changes are not propagated to the datastore until commit()/flush(). The data is checked just before commit to ensure the integrity in this respect. The most convenient way of checking data for updates is to maintain a column on each table that handles optimistic transaction data. The user will decide this when generating their MetaData.

Rather than placing version/timestamp columns on all user datastore tables, JPA1 allows the user to notate particular classes as requiring optimistic treatment. This is performed by specifying in MetaData or annotations the details of the field/column to use for storing the version - see versioning for JPA. With JPA1 you must have a field in your class ready to store the version.

In JPA1 you can read the version by inspecting the field marked as storing the version value.

In terms of the process of an optimistic transaction, we demonstrate this below.

Operation	DataNucleus process	Datastore process
Start transaction		

Operation	DataNucleus process	Datastore process
Persist object	Prepare object (1) for persistence	
Update object	Prepare object (2) for update	
Persist object	Prepare object (3) for persistence	
Update object	Prepare object (4) for update	
Flush	Flush all outstanding changes to the datastore	Open connection. Version check of object (1) Insert the object (1) in the datastore. Version check of object (2) Update the object (2) in the datastore. Version check of object (3) Insert the object (3) in the datastore. Version check of object (4) Update the object (4) in the datastore.
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	
Update object	Prepare object (6) for update	
Commit transaction	Flush all outstanding changes to the datastore	Version check of object (5) Insert the object (5) in the datastore Version check of object (6) Update the object (6) in the datastore. Commit connection.

Here no changes make it to the datastore until the user either commits the transaction, or they invoke flush(). The impact of this is that when performing a query, by default, the results may not contain the modified objects unless they are flushed to the datastore before invoking the query. Depending on whether you need the modified objects to be reflected in the results of the query governs what you do about that. If you invoke flush() just before running the query the query results will include the changes. The obvious benefit of optimistic transaction is that all changes are made in a block and version checking of objects is performed before application of changes, hence this mode copes better with external processes updating the objects.

See also :-

- JPA MetaData reference for <version> element
- JPA Annotations reference for @Version

No Transactions

DataNucleus allows the ability to operate without transactions. With JPA this is the default behaviour. DataNucleus supports these allowing the ability to read objects and make updates outside of transactions. The important thing is that to use this mode of operation, you must enable it by setting the property (DataNucleus defaults to false if not set). Please be aware that any non transactional updates will typically be committed to the datastore **by the subsequent transaction**, with the updates being queued until that point.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

11.7 Transactions

JPA Transactions

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. Transactions can be managed by the users application, or can be managed by a framework (such as Spring), or can be managed by a J2EE container. These are described below.

See also :-

- Transaction Types

Locally Managed Transactions

When using a JPA implementation such as DataNucleus in a J2SE environment, the Transactions are known as Locally Managed Transactions. The users code will manage the transactions by starting, and commiting the transaction itself. With these transactions with JPA Javadoc you would do something like

```
EntityManager em = emf.getEntityManager();
EntityTransaction tx = em.getTransaction();
try
{
    tx.begin();

    {users code to persist objects}

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
em.close();
```

When you use a framework like Spring you would not need to specify the tx.begin(), tx.commit(), tx.rollback() since that would be done for you. The basic idea with Locally Managed transactions is that you are managing the transaction start and end.

Container Managed Transactions

When using a J2EE container you are giving over control of the transactions to the container. Here you have Container Managed Transactions. In terms of your code, you would do like the previous example except that you would OMIT the tx.begin(), tx.commit(), tx.rollback() since the J2EE container will be doing this for you.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Transaction Isolation

DataNucleus also allows specification of the transaction isolation level. This is specified via the EntityManagerFactory property datanucleus.transactionIsolation. It accepts the standard JDBC values of

- **read-uncommitted (1)** : dirty reads, non-repeatable reads and phantom reads can occur
- **read-committed (2)** : dirty reads are prevented; non-repeatable reads and phantom reads can occur
- **repeatable-read (4)** : dirty reads and non-repeatable reads are prevented; phantom reads can occur
- **serializable (8)** : dirty reads, non-repeatable reads and phantom reads are prevented

The default is read-committed. If the datastore doesn't support a particular isolation level then it will silently be changed to one that is supported (for example DB4O only supports *read-committed* so will always use that). As an alternative you can also specify it on a per-transaction basis as follows (using the values in parentheses above).

```
org.datanucleus.jpa.EntityTransactionImpl tx =
(org.datanucleus.jpa.EntityTransactionImpl)pm.currentTransaction();
tx.setOption("transaction.isolation", 2);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.8 Cache

JPA Caching

Caching is an essential mechanism in providing efficient usage of resources in many systems. Caching allows objects to be retained and returned rapidly without having to make an extra call to the datastore. JPA1 defines very little about caching, but the JPA2 specification should improve on this. DataNucleus provides a definition of caching at 2 levels. The 2 levels of caching available are

- Level 1 Cache - represents the caching of instances within an EntityManager
- Level 2 Cache - represents the caching of instances within an EntityManagerFactory (across multiple EntityManager's)

You can think of a cache as a Map, with values referred to by keys. In the case of JPA, the key is the object identity (identity is unique in JPA).

Level 1 Cache

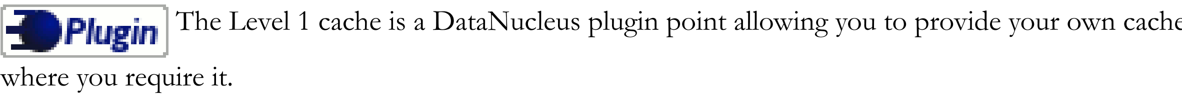
The Level 1 Cache is always enabled. There are inbuilt types for the Level 1 Cache available for selection.

DataNucleus supports the following types of Level 1 Cache.

- weak - uses a weak reference backing map. If JVM garbage collection clears the reference, then the object is removed from the cache. This is the default Level 1 Cache
- soft - uses a soft reference backing map. If the map entry value object is not being actively used, then garbage collection may garbage collect the reference, in which case the object is removed from the cache.
- hard - uses a normal HashMap backing. With this option all references are strong meaning that objects stay in the cache until they are explicitly removed by calling remove() on the cache.

You can specify the type of Level 1 Cache by providing the PMF property datanucleus.cache.level1.type. You set this to the value of the type required. If you want to remove all objects from the L1 cache programmatically you should use em.clear() but bear in mind the other things that this does.

Objects are placed in the L1 cache (and updated there) during the course of the transaction. This provides rapid access to the objects in use in the users application and is used to guarantee that there is only one object with a particular identity at any one time for that EntityManager.



The Level 1 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it.

Level 2 Cache

By default in DataNucleus, Level 2 Cache is disabled. The user can configure the Level 2 Cache if they so wish. This is controlled by use of the persistence property datanucleus.cache.level2 which is a boolean property. There are builtin Level 2 caches available for use. You can specify the type of Level 2 Cache by providing the persistence property datanucleus.cache.level2.type. You set this to type of cache required.

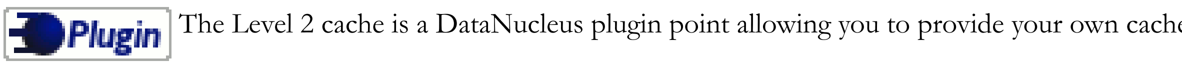
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

With the Level 2 Cache you currently have the following options.

- default - the default option. Provides support for the JPA2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Weak references are held to non pinned objects.
- soft - Provides support for the JPA2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Soft references are held to non pinned objects.
- EHCache - a simple wrapper to EHCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- EHCacheClassBased - similar to the EHCache option but class-based.
- OSCache - a simple wrapper to OSCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- SwarmCache - a simple wrapper to SwarmCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- Oracle Coherence - a simple wrapper to Oracle's Coherence caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning. Oracle's caches support distributed caching, so you could, in principle, use DataNucleus in a distributed environment with this option.

The EHCache, OSCache, SwarmCache and Coherence caches are available in the datanucleus-cache plugin.

From Access Platform 1.0 M3 onwards objects are placed in the L2 cache when you commit() the transaction of a EntityManager. This means that you only have datastore-persisted objects in that cache. Also, if an object is deleted during a transaction then at commit it will be removed from the L2 cache if it is present.



The Level 2 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it. Use the examples of the EHCache, Coherence caches etc as reference.

Controlling the Level 2 Cache

The majority of times when using a JPA-enabled system you will not have to take control over any aspect of the caching other than specification of whether to use a Level 2 Cache or not. With JPA2 and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the EntityManagerFactory.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(persUnitName,
props);
Cache cache = emf.getCache();
```

The Cache interface provides methods to control the retention of objects in the cache. You have 2 types of methods

- contains - check if an object of a type with a particular identity is in the cache
- evict - used to remove objects from the Level 2 Cache

From Access Platform 1.0 M3 onwards you can also control which classes are put into a Level 2 cache. You do this by specifying the extension **cacheable** set to *false*. So with the following specification, no objects of type *MyClass* will be put in the L2 cache.

```
@Extension(vendorName="datanucleus", key="cacheable", value="false")
@Entity
public class MyClass
{
    ...
}
```

From Access Platform 1.0 M3 onwards you can also control which fields of an objects are put in the Level 2 cache. You do this by specifying the extension **cacheable** for the field in question setting it to either *true* or *false*. The default is *true*. This setting is only required for fields that are relationships to other persistable objects. Like this

```
public class MyClass
{
    ...

    Collection values;

    @Extension(vendorName="datanucleus", key="cacheable", value="false")
    Collection elements;
}
```

So in this example we will cache "values" but not "elements". If a field is *cacheable* then

- If it is a persistable object, the "identity" of the related object will be stored in the Level 2 cache for this field of this object
- If it is a Collection of persistable elements, the "identity" of the elements will be stored in the Level 2 cache for this field of this object
- If it is a Map of persistable keys/values, the "identity" of the keys/values will be stored in the Level 2 cache for this field of this object

When pulling an object in from the Level 2 cache and it has a reference to another object Access Platform uses the "identity" to find that object in the Level 1 or Level 2 caches to re-relate the objects.

L2 Cache using Oracle Coherence

DataNucleus provides a simple wrapper to Oracle's Coherence caches. This currently takes the NamedCache interface in Coherence and instantiates a cache of a user provided name. To enabled this you should set the following persistence properties

```
    datanucleus.cache.level2=true
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    datanucleus.cache.level2.type=coherence
    datanucleus.cache.level2.cacheName={coherence cache name}
```

The Coherence cache name is the name that you would normally put into a call to CacheFactory.getCache(name). You have the benefits of Coherence's distributed/serialized caching. If you require more control over the Coherence cache whilst using it with DataNucleus, you can just access the cache directly via

```
    DataStoreCache cache = pmf.getDataStoreCache();
    NamedCache tangosolCache = ((TangosolLevel2Cache)cache).getTangosolCache();
```

L2 Cache using EHCache

DataNucleus provides a simple wrapper to EHCache's caches. To enable this you should set the persistence properties

```
    datanucleus.cache.level2=true
    datanucleus.cache.level2.type=ehcache
    datanucleus.cache.level2.cacheName={cache name}
    datanucleus.cache.level2.configurationFile={EHCache configuration file (in
classpath)}
```

The EHCache plugin also provides an alternative L2 Cache that is class-based. To use this you would need to replace "ehcache" above with "ehcacheclassbased".

L2 Cache using OSCache

DataNucleus provides a simple wrapper to OSCache's caches. To enable this you should set the persistence properties

```
    datanucleus.cache.level2=true
    datanucleus.cache.level2.type=oscache
    datanucleus.cache.level2.cacheName={cache name}
```

L2 Cache using SwarmCache

DataNucleus provides a simple wrapper to SwarmCache's caches. To enable this you should set the persistence properties

```
    datanucleus.cache.level2=true
    datanucleus.cache.level2.type=swarmcache
    datanucleus.cache.level2.cacheName={cache name}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.9 Lifecycle Callbacks

Lifecycle Callbacks

JPA1 defines a mechanism whereby an Entity can be marked as a listener for lifecycle events. Alternatively a separate entity listener class can be defined to receive these events. Thereafter when entities of the particular class go through lifecycle changes events are passed to the provided methods. Let's look at the two different mechanisms

Entity Callbacks

An Entity itself can have several methods defined to receive events when any instances of that class pass through lifecycles changes. Let's take an example

```
@Entity
public class Account
{
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }

    @PrePersist
    protected void validateCreate()
    {
        if (getBalance() < MIN_REQUIRED_BALANCE)
        {
            throw new AccountException("Insufficient balance to open an account");
        }
    }

    @PostLoad
    protected void adjustPreferredStatus()
    {
        preferred = (getBalance() >= AccountManager.getPreferredStatusLevel());
    }
}
```

So in this example just before any "Account" object is persisted the *validateCreate* method will be called. In the same way, just after the fields of any "Account" object are loaded the *adjustPreferredStatus* method is called. Very simple.

You can register callbacks for the following lifecycle events

- PrePersist
- PostPersist

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- PreRemove
- PostRemove
- PreUpdate
- PostUpdate
- PostLoad

The only other rule is that any method marked to be a callback method has to take no arguments as input, and have void return.

Entity Listener

As an alternative to having the actual callback methods in the Entity class itself you can define a separate class as an *EntityListener*. So lets take the example shown before and do it for an EntityListener.

```
@Entity
@EntityListeners(org.datanucleus.MyEntityListener.class)
public class Account
{
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }
}
```

```
public class MyEntityListener
{
    @PostPersist
    public void newAccountAlert(Account acct)
    {
         ... do something when we get a new Account
    }
}
```

So we define our "Account" entity as normal but mark it with an EntityListener, and then in the *EntityListener* we define the callbacks we require. As before we can define any of the 7 callbacks as we require. The only difference is that the callback method has to take an argument of type "Object" that it will be called for, and have void return.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

12.1 REST API

REST API

The REST API provides a RESTful JSON interface to the datastore. All entities are accessed, queried and stored as resources via well defined HTTP methods.

Support for REST is in its early stages and will be enhanced in future releases.

Usage

The REST API automatically exposes the persistent class in RESTful style, and requires minimum configuration as detailed in the sections linked below.

- With REST you need to configure the Servlet to enable the HTTP interface
- With REST you need to use the HTTP methods to persist or access objects

Known Limitations

- Only Persistent class using Application Identity are supported.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

12.2 Configuration

Configuration

The configuration of the REST API consists in the deployment of jar libraries to the classpath and the configuration of the servlet in the */WEB-INF/web.xml*. After it's configured, all persistent classes are automatically exposed via RESTful HTTP interface.

Libraries

DataNucleus REST API requires the libraries: DataNucleus core, DataNucleus rest, Flexjson, and the JDO API. These libraries must be in the classpath. In war files, these libraries are deployed under the folder */WEB-INF/lib/*.

web.xml

The DataNucleus REST Servlet class implementation is *org.datanucleus.rest.RestServlet*. It has to be configured in the */WEB-INF/web.xml* file, and it takes one initialisation parameter.

Parameter	Description
persistence-context	A named PMF/persistence-unit configuration

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">

    <servlet>
        <servlet-name>DataNucleus</servlet-name>
        <servlet-class>org.datanucleus.rest.RestServlet</servlet-class>
        <init-param>
            <param-name>persistence-context</param-name>
            <param-value>transactions-optional</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>DataNucleus</servlet-name>
        <url-pattern>/dn/*</url-pattern>
    </servlet-mapping>

...
</web-app>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

12.3 HTTP Methods

HTTP Methods

The persistence to the datastore is performed via HTTP methods as following:

Method	Operation	URL format	Return	Arguments
GET	Fetch all objects for a class	/{full-class-name}	JSON Array of JSON objects	
GET	Fetch a single object.	/{full-class-name}/{primary key}	A JSON object	The primary key field is passed in the URL path
GET	Query objects. Returns a JSON Array of objects	/{full-class-name}?{filter}	JSON Array of JSON objects	A JDO/JPA filter query
PUT	Update objects	/{full-class-name}/{primary key}	The JSON object is returned. If using application identity, with the primary key is included.	The primary key field is passed in the URL path; The JSON Object is passed in the HTTP Content
POST	Insert objects	/{full-class-name}	The JSON object is returned. If using application identity, with the primary key is included.	The JSON Object is passed in the HTTP Content
DELETE	delete object	/{full-class-name}/{primary key}		The primary key field is passed in the URL path

HTTP Methods samples

Fetch all objects of class guestbook.Greeting

```
GET http://datanucleus.appspot.com/dn/guestbook.Greeting
```

Response:

```
[{"author":null,
  "class":"guestbook.Greeting",
  "content":"test",
  "date":1239213624216,
  "id":1},
 {"author":null,
  "class":"guestbook.Greeting",
  "content":"test2",
  "date":1239213632286,
  "id":2}]
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Fetch object with id 1 of class guestbook.Greeting

```
GET http://datanucleus.appspot.com/dn/guestbook.Greeting/1
```

Response:

```
{"author":null,
  "class":"guestbook.Greeting",
  "content":"test",
  "date":1239213624216,
  "id":1}
```

Fetch object using Application Primary Key Class

```
GET
http://dnsamplemaps.appspot.com/dn/google.maps.Markers/{"class":"com.google.appengine.api.datastore.Key",
```

Response:

```
{"class":"google.maps.Markers",
 "key":{"class":"com.google.appengine.api.datastore.Key",
        "id":1001,
        "kind":"Markers"
       },
 "markers":[
            {"class":"google.maps.Marker",
             "html":"Paris",
             "key":{"class":"com.google.appengine.api.datastore.Key",
                    "id":1,
                    "kind":"Marker",
                    "parent":{"class":"com.google.appengine.api.datastore.Key",
                              "id":1001,
                              "kind":"Markers"
                             }
                   },
             "lat":48.862222,
             "lng":2.351111
            }
           ]
}
```

Query object of class guestbook.Greeting

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
GET http://datanucleus.appspot.com/dn/guestbook.Greeting?content=='test'
```

Response:

```
[{"author":null,
  "class":"guestbook.Greeting",
  "content":"test",
  "date":1239213624216,
  "id":1}]
```

Insert a new object of class guestbook.Greeting

```
POST http://datanucleus.appspot.com/dn/guestbook.Greeting
{"author":null,
  "class":"guestbook.Greeting",
  "content":"test insert",
  "date":1239213923232}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.1 JPA Query API

JPA Query API

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JPA specifies support for a semi-object-oriented query language (JPQL) and a relational query language (SQL). DataNucleus supports the following

- JPQL - language based around the objects that are persisted **but** with syntax very similar to SQL
- SQL - language found on almost all RDBMS, which JPA is targetted at.

Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer JPQL. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of SQL. This is the power of an implementation like DataNucleus in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

There are 2 categories of queries with JPA :-

- Named Query where the query is defined in MetaData and referred to by its name at runtime.
- **Programmatic Query** where the query is defined using the JPA1 Query API.

Let's now try to understand the Query API in JPA Javadoc, We firstly need to look at a typical Query. We'll take 2 examples

JPQL Query

Let's create a JPQL query to highlight its usage

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold
ORDER BY p.param1 ascending");
query.setParameter("threshold", my_threshold);
List results = query.getResultList();
```

In this Query, we implicitly select JPQL by using the method *EntityManager.createQuery()*, and the query is specified to return all objects of type Product (or subclasses) which have the field param2 less than some threshold value ordering the results by the value of field param1. We've specified the query like this because we want to pass the threshold value in as a parameter (so maybe running it once with one value, and once with a different value). We then set the parameter value of our threshold parameter. The Query is then executed to return a List of results. The example is to highlight the typical methods specified for a (JPQL) Query.

SQL Query

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Let's create an SQL query to highlight its usage

```
Query query = em.createNativeQuery("SELECT * FROM Product p WHERE p.param2 < ?1");
query.setParameter(1, my_threshold);
List results = query.getResultList();
```

So we implicitly select SQL by using the method *EntityManager.createNativeQuery()*, and the query is specified like in the JPQL case to return all instances of type *Product* (using the table name in this SQL query) where the column *param2* is less than some threshold value.

setFirstResult(), setMaxResults()

In JPA to specify the range of a query you have two methods available. So you could do

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold
ORDER BY p.param1 ascending");
query.setFirstResult(1);
query.setMaxResults(3);
```

so we will get results 1, 2, and 3 returned only. The first result starts at 0 by default.

setHint()

JPA's query API allows implementations to support extensions ("hints") and provides a simple interface for enabling the use of such extensions on queries.

```
q.setHint("extension_name", value);
```

DataNucleus provides various extensions for different types of queries.

setParameter()

JPA's query API supports named and numbered parameters and provides method for setting the value of particular parameters. To set a named parameter, for example, you could do

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold
ORDER BY p.param1 ascending");
q.setParameter("threshold", value);
```

To set a numbered parameter you could do

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < ?1 ORDER BY
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
p.param1 ascending");
q.setParameter(1, value);
```

Numbered parameters are numbered from 1.

getResultList()

To execute a JPA query you would typically call *getResultList*. This will return a List of results. This should not be called when the query is an "UPDATE"/"DELETE".

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold
ORDER BY p.param1 ascending");
q.setParameter("threshold", value);
List results = q.getResultList();
```

getSingleResult()

To execute a JPA query where you are expecting a single value to be returned you would call *getSingleResult*. This will return the single Object. If the query returns more than one result then you will get an Exception. This should not be called when the query is an "UPDATE"/"DELETE".

```
Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 = :value");
q.setParameter("value", val1);
Product prod = q.getSingleResult();
```

executeUpdate()

To execute a JPA UPDATE/DELETE query you would call *executeUpdate*. This will return the number of objects changed by the call. This should not be called when the query is a "SELECT".

```
Query query = em.createQuery("DELETE FROM Product p");
int number = q.executeUpdate();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.2 Named Queries

JPA Named Queries

With the JPA1 API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called Product (something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```
<entity class="Product">
    ...
    <named-query name="SoldOut"><![CDATA[
    SELECT p FROM Product p WHERE p.status == "Sold Out"
    ]]></named-query>
</entity>
```

So we have a JPQL query called "SoldOut" defined for the class Product that returns all Products (and subclasses) that have a status of "Sold Out". Out of interest, what we would then do in our application to execute this query woule be

```
Query query = em.createNamedQuery("SoldOut");
List results = query.getResultList();
```

The above example was for the JPQL query language. We can do a similar thing using SQL, so we define the following in our MetaData for our Product class

```
<entity class="Product">
    ...
    <named-native-query name="PriceBelowValue"><![CDATA[
    SELECT NAME FROM PRODUCT WHERE PRICE < ?1
    ]]></named-native-query>
</entity>
```

So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named query, asking for the names of all Products with price below 20 euros.

```
Query query = em.createNamedNativeQuery("PriceBelowValue");
query.setParameter(1, new Double(20.0));
List results = query.getResultList();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

All of the examples above have been specifed within the <entity> element of the MetaData. You can also define these named queries in annotations in the class itself, but clearly that is polluting your model.

Please proceed to the sections specific to JPQL and SQL for details on the precise nature of the query for the querying languages supported by DataNucleus with JPA.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.3 JPQL

JPQL SELECT Queries

The JPA specification defines JPQL, for selecting objects from the datastore. To provide a simple example, this is what you would do

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = 'Jones'");
List results = (List)q.getResultsList();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

SELECT Syntax

In JPQL queries you define the query in a single string, defining the result, the candidate class(es), the filter, any grouping, and the ordering. This string has to follow the following pattern

```
SELECT [<result>]
    [FROM <candidate-class(es)>]
    [WHERE <filter>]
    [GROUP BY <grouping>]
    [HAVING <having>]
    [ORDER BY <ordering>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

Entity Name

In the example shown you note that we did not specify the full class name. We used *Person p* and thereafter could refer to *p* as the alias. The *Person* is called the **entity name** and in JPA MetaData this can be defined against each class in its definition. For example

```
<entity class="org.datanucleus.company.Person" name="Person">
    ...
</entity>
```

Input Parameters

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Let's take two examples

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Named Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :surname AND
o.firstName = :forename");
q.setParameter("surname", theSurname);
q.setParameter("forename", theForename");

Numbered Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND
p.firstName = ?2");
q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

So in the first case we have parameters that are prefixed by **:** (colon) to identify them as a parameter and we use that name when calling *Query.setParameter()*. In the second case we have parameters that are prefixed by **?** (question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*.

Range of Results

With JPQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.age &gt; 20");
q.setFirstResult(0);
q.setMaxResults(20);
```

So with this query we get results 0 to 19 inclusive.

Query Execution

There are two ways to execute a JPQL query. When you know it will return 0 or 1 results you call

```
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don't know then you would call

```
List results = query.getResultList();
```

JPQL DELETE Queries

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The JPA specification defines a mode of JPQL for deleting objects from the datastore. DataNucleus AccessPlatform supports these for db4o, Excel, LDAP, NeoDatis and XML datastores.

DELETE Syntax

The syntax for deleting records is very similar to selecting them

```
DELETE FROM [<candidate-class>]
    [WHERE <filter>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

JPQL UPDATE Queries

The JPA specification defines a mode of JPQL for updating objects in the datastore. DataNucleus doesnt currently support this but will soon.

UPDATE Syntax

The syntax for updating records is very similar to selecting them

```
UPDATE [<candidate-class>] SET item1=value1, item2=value2
    [WHERE <filter>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.4 JPQL : Subqueries

JPQL Subqueries

With JPQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JPQL also allows the use of subqueries. Here's an example

```
SELECT Object(e) FROM org.datanucleus.Employee e
WHERE e.salary > (SELECT avg(f.salary) FROM org.datanucleus.Employee f)
```

So we want to find all Employees that have a salary greater than the average salary. The subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "f", whereas in the outer query the alias is "e".

All or Any Expressions

One use of subqueries with JPQL is where you want to compare with some or all of a particular expression. To give an example

```
SELECT emp FROM Employee emp
WHERE emp.salary > ALL (SELECT m.salary FROM Manager m WHERE m.department =
emp.department)
```

So this returns all employees that earn more than all managers in the same department! You can also compare with some/any, like this

```
SELECT emp FROM Employee emp
WHERE emp.salary > ANY (SELECT m.salary FROM Manager m WHERE m.department =
emp.department)
```

So this returns all employees that earn more than any one Manager in the same department.

Existence Expressions

Another use of subqueries in JPQL is where you want to check on the existence of a particular thing. For example

```
SELECT DISTINCT emp FROM Employee emp
WHERE EXISTS (SELECT emp2 FROM Employee emp2 WHERE emp2 = emp.spouse)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So this returns the employees that have a partner also employed.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.5 JPQL : In-Memory

JPQL : In-Memory queries

The typical use of a JPQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JPQL syntax in the datastore *native query language* and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of *in-memory evaluation* in DataNucleus.

- Query methods **CONTAINS(Collection)**, **CONTAINS(Map)** are not currently supported
- Subqueries using ALL, ANY, SOME, EXISTS are not currently supported
- MEMBER OF syntax is not currently supported.

To enable evaluation in memory you specify the query hint **datanucleus.query.evaluateInMemory** to *true* as follows

```
query.setHint("datanucleus.query.evaluateInMemory","true");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

13.6 SQL

SQL Queries

The JPA specification defines its interpretation of SQL, for selecting objects from the datastore. To provide a simple example, this is what you would do

```
Query q = em.createNativeQuery("SELECT p.id, o.firstName, o.lastName FROM Person p,
Job j " +
                "WHERE (p.job = j.id) AND j.name = 'Cleaner'");
List results = (List)q.getResultsList();
```

This finds all "Person" objects that do the job of "Cleaner". The syntax chosen has to be runnable on the RDBMS that you are using (and since SQL is anything but "standard" you will likely have to change your query when moving to another datastore).

Input Parameters

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Here's an example

```
Numbered Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND
p.firstName = ?2");
q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

So we have parameters that are prefixed by **?** (question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*. With SQL queries we can't use named parameters. This is known as *numbered* parameters.

DataNucleus also supports use of *named* parameters where you assign names just like in JPQL. This is not defined by the JPA specification so dont expect other JPA implementations to support it. Let's take the previous example and rewrite it using *named* parameters, like this

```
Named Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :firstParam AND
p.firstName = :otherParam");
q.setParameter("firstParam", theSurname);
q.setParameter("otherParam", theForename);
```

Range of Results

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

With SQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```
Query q = em.createNativeQuery("SELECT p FROM Person p WHERE p.age &gt; 20");
q.setFirstResult(0);
q.setMaxResults(20);
```

So with this query we get results 0 to 19 inclusive.

Query Execution

There are two ways to execute a SQL query. When you know it will return 0 or 1 results you call

```
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don't know then you would call

```
List results = query.getResultList();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.1 Datastores

Datastores

The DataNucleus Access Platform is designed for flexibility to operate with any type of datastore. We already support a very wide range of datastores and this will only increase in the future. In this section you can find the specifics for particular supported datastores over and above what was already addressed for JDO and JPA persistence.

- RDBMS : tried and tested since the 1970s, relational databases form an integral component of many systems. They incorporate optimised querying mechanisms, yet also can suffer from object-relational impedance mismatch in some situations. They also require an extra level of configuration to map from objects across to relational tables/columns.
- DB4O : an open source object datastore. This provides fast persistence of large object graphs, without the necessity of any object-relational mapping.
- NeoDatis : an open source object datastore. This provides fast persistence of large object graphs, without the necessity of any object-relational mapping.
- LDAP : an internet standard datastore for indexed data that is not changing significantly.
- Excel : Excel spreadsheets provide a widely used format allowing publishing of data, making it available to other groups.
- XML : XML defines a document format and, as such, is a key data transfer medium.
- JSON : another format of document for exchange, in this case with particular reference to web contexts.
- Open Document Format (ODF) : ODF is an international standard document format, and its spreadsheets provide a widely used form for publishing of data, making it available to other groups.
- Google AppEngine : AppEngine provides persistence to its own *BigTable* datastore using a DataNucleus plugin.

If you have a requirement for persistence to some other datastore, then it would likely be easily provided by creation of a DataNucleus *StoreManager*. Please contact us via the forum so that you can provide this and contribute it back to the community.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2 RDBMS

Supported RDBMS Datastores

DataNucleus supports persisting objects to RDBMS datastores (using the datanucleus-rdbms plugin). It supports the vast majority of RDBMS products available today. DataNucleus communicates with the RDBMS datastore using JDBC. RDBMS systems accept varying standards of SQL and so DataNucleus will support particular RDBMS/JDBC combinations only, though clearly we try to support as many as possible.

By default when you create a PersistenceManagerFactory (PMF) to connect to a particular datastore DataNucleus will automatically detect the datastore adapter to use and will use its own internal adapter for that type of datastore. If you find that either DataNucleus has incorrectly detected the adapter to use, or that there is some issue with the internal adapter, you can override the default behaviour. Please refer to the Database Adapter Extension Guide for details.

The table below shows the versions of RDBMS and JDBC driver that DataNucleus has been tested with

Compatibility

RDBMS Name	RDBMS Version	JDBC Name	JDBC Version
	3.23	mysql-connector-java	3.0
	4.0	mysql-connector-java	3.0.16, 3.1.12
	4.1	mysql-connector-java	3.0.16, 3.1.12
	5.0	mysql-connector-java	3.1.12
MS SQL Server	2000	Windows	Microsoft SQL Server JDBC Driver
	2005	Windows	Microsoft SQL Server JDBC Driver
	8	JDBC	ojdbc14.jar and/or classes12
	9	JDBC	ojdbc14.jar and/or classes12
	10.2	JDBC	ojdbc14.jar (Oracle JDBC driver 10.2)
	12.5	JConnect	5.5
	15.0	JConnect	6.0.5
	1.7	HSQLDB	1.7
	1.8	HSQLDB	1.8
	1.0	H2	1.0

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

RDBMS Name	RDBMS Version	JDBC Name	JDBC Version
McKoi	1.0.3	Mckoi JDBC Driver	1.0
PostgreSQL POWERED	7.3	PostgreSQL	7.3
	7.4	PostgreSQL	7.4
	8.0	PostgreSQL	8.0
	8.1	PostgreSQL	8.1
	8.2	PostgreSQL	8.1
PostGIS Geospatial Objects for PostgreSQL	1.1.3 (+ PostgreSQL 8.1)	PostGIS + PostgreSQL	1.1.6, 1.2.0
	1.1.4 (+ PostgreSQL 8.1)	PostGIS + PostgreSQL	1.1.6, 1.2.0
	1.1.5 (+ PostgreSQL 8.1)	PostGIS + PostgreSQL	1.1.6, 1.2.0
	1.1.6 (+ PostgreSQL 8.1)	PostGIS + PostgreSQL	1.1.6, 1.2.0
	1.2.0 (+ PostgreSQL 8.1)	PostGIS + PostgreSQL	1.1.6, 1.2.0
Pointbase			
Apache Derby	10.0	Derby	10.0
	10.1	Derby	10.1
	10.2	Derby	10.2
DB2	08.01	IBM DB2 JDBC 2.0 Type 2	08.01
	08.02	IBM DB2 JDBC 2.0 Type 2	08.02
	DB2 UDB for AS/400 v5.2	AS/400 Toolbox for Java JDBC Driver	05.02
Firebird	1.5.1	Firebird	1.5.0
SAPDB / MaxDB	7.6.0	MaxDB JDBC Driver	7.6.0
Informix	11.x	Informix JDBC	3.0

If you have success with any other combinations for the above RDBMS or indeed with any other RDBMS, please let us know so we can update our compatibility guide. We only show here what we have either tried ourselves or what has been reported as successful.

DB2

To specify DB2 as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.ConnectionDriverName=COM.ibm.db2.jdbc.app.DB2Driver
datanucleus.ConnectionURL=jdbc:db2:'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'

#DB2network driver
#datanucleus.ConnectionDriverName=COM.ibm.db2.jdbc.net.DB2Driver
#datanucleus.ConnectionURL=jdbc:db2://hostname:port//dbname
```

MySQL

MySQL is supported as an RDBMS datastore by DataNucleus with the following provisos

- INNODB tables must be used since it is the only table type that allows foreign keys etc at the moment. You can however define what type your table uses by setting the <class> extension "mysql-engine-type" to be MyISAM or whatever for the class being persisted.
- JDOQL.isEmpty()/contains() will not work in MySQL 4.0 (or earlier) since the query uses EXISTS and that is only available from MySQL 4.1
- MySQL on Windows MUST specify *datanucleus.identifier.case* as "LowerCase" since the MySQL server stores all identifiers in lowercase BUT the mysql-connector-java JDBC driver has a bug (in versions up to and including 3.1.10) where it claims that the MySQL server stores things in mixed case when it doesnt
- MySQL 3.* will not work reliably with inheritance cases since DataNucleus requires UNION and this doesn't exist in MySQL 3.*
- MySQL before version 4.1 will not work correctly on JDOQL Collection.size(), Map.size() operations since this requires subqueries, which are not supported before MySQL 4.1.
- If you receive an error "Incorrect arguments to mysql_stmt_execute" then this is a bug in MySQL and you need to update your JDBC URL to append "?useServerPrepStmts=false".
- MySQL throws away the milliseconds on a Date and so cannot be used reliably for Optimistic locking using strategy "date-time" (use "version" instead)
- You can specify "BLOB", "CLOB" JDBC types when using MySQL with DataNucleus but you must turn validation of columns OFF. This is because these types are not supported by the MySQL JDBC driver and it returns them as LONGVARBINARY/LONGVARCHAR when querying the column type

To specify MySQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.mysql.jdbc.Driver
datanucleus.ConnectionURL=jdbc:mysql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

MS SQL Server

MS SQL Server is supported as an RDBMS datastore by DataNucleus with the following proviso

- MS SQL 2000 does not keep accuracy on *datetime* datatypes. This is an MS SQL 2000 issue. In order to keep the accuracy when storing *java.util.Date* java types, use *int* datatype.

To specify MS SQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

Microsoft SqlServer 2005 JDBC Driver (Recommended)

```
datanucleus.ConnectionDriverName=com.microsoft.sqlserver.jdbc.SQLServerDriver
datanucleus.ConnectionURL=jdbc:sqlserver://'host':'port';DatabaseName='db-name'
                     ;SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Microsoft SqlServer 2000 JDBC Driver

```
datanucleus.ConnectionDriverName=com.microsoft.jdbc.sqlserver.SQLServerDriver
datanucleus.ConnectionURL=jdbc:microsoft:sqlserver://'host':'port';DatabaseName='db-name'
                     ;SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Oracle

To specify Oracle as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc) ... you can also use 'oci' instead of 'thin' depending on your driver.

```
datanucleus.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
datanucleus.ConnectionURL=jdbc:oracle:thin:@'host':'port':'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Sybase

To specify Sybase as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.ConnectionDriverName=com.sybase.jdbc2.jdbc.SybDriver
datanucleus.ConnectionURL=jdbc:sybase:Tds:'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

HSQLDB

HSQLDB is supported as an RDBMS datastore by DataNucleus with the following proviso

- Use of batched statements is disabled since HSQLDB has a bug where it throws exceptions "batch failed" (really informative). Still waiting for this to be fixed in HSQLDB
- Use of JDOQL/JPQL subqueries cannot be used where you want to refer back to the parent query since HSQLDB up to and including version 1.8 don't support this.

To specify HSQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.hsqldb.jdbcDriver
datanucleus.ConnectionURL=jdbc:hsqldb:hsql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

H2

H2 is supported as an RDBMS datastore by DataNucleus

To specify H2 as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.h2.Driver
datanucleus.ConnectionURL=jdbc:h2:'db-name'
datanucleus.ConnectionUserName=sa
datanucleus.ConnectionPassword=
```

Informix

Informix is supported as an RDBMS datastore by DataNucleus

To specify Informix as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.informix.jdbc.IfxDriver
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.ConnectionURL=jdbc:informix-sqli://[{ip|host}:port][/dbname]:
                    INFORMIXSERVER=servername[;name=value[;name=value]...]
datanucleus.ConnectionUserName=informix
datanucleus.ConnectionPassword=password
```

e.g.

```
datanucleus.ConnectionDriverName=com.informix.jdbc.IfxDriver
datanucleus.ConnectionURL=jdbc:informix-sqli://192.168.254.129:9088:
                    informixserver=demo_on;database=buf_log_db
datanucleus.ConnectionUserName=informix
datanucleus.ConnectionPassword=password
```

Note that some database logging options in Informix do not allow changing autoCommit dinamically. You need to rebuild the database to support it. To rebuild the database refer to Informix documention, but as example, run $INFORMIXDIR\bin\dbaccess and execute the command "CREATE DATABASE mydb WITH BUFFERED LOG".

INDEXOF: Informix 11.x does not have a function to search a string in another string. DataNucleus defines a user defined function, DATANUCLEUS_STRPOS, which is automatically created on startup. The SQL for the UDF function is:

```
create function DATANUCLEUS_STRPOS(str char(40),search char(40),from smallint)
returning smallint
    define i,pos,lenstr,lensearch smallint;
    let lensearch = length(search);
    let lenstr = length(str);

    if lenstr=0 or lensearch=0 then return 0; end if;

    let pos=-1;
    for i=1+from to lenstr
        if substr(str,i,lensearch)=search then
            let pos=i;
            exit for;
        end if;
    end for;
    return pos;
end function;
```

McKoi

McKoi is supported as an RDBMS datastore by DataNucleus with the following proviso

- McKoi doesn't provide full information to allow correct validation of tables/constraints.

To specify McKoi as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.mckoi.JDBCDriver
datanucleus.ConnectionURL=jdbc:mckoi://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

PostgreSQL

To specify PostgreSQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

PostgreSQL with PostGIS extension

To specify PostGIS as your datastore, you will need to decide first which geometry library you want to use and then set the connection url accordingly.

For the PostGIS JDBC geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For Oracle's JGeometry you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jgeom://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For the JTS (Java Topology Suite) geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jts://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Apache Derby/Cloudscape

Apache Derby is supported as an RDBMS datastore by DataNucleus

To specify Apache Derby/Cloudscape as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.apache.derby.jdbc.EmbeddedDriver
datanucleus.ConnectionURL=jdbc:derby:'db-name';create=true
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Above settings are used together with the Apache Derby in embedded mode. The below settings are used in network mode, where the default port number is 1527.

```
datanucleus.ConnectionDriverName=org.apache.derby.jdbc.ClientDriver
datanucleus.ConnectionURL=jdbc:derby://'hostname':'portnumber'/'db-name';create=true
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

org.apache.derby.jdbc.ClientDriver

ASCII: Derby 10.1 does not have a function to convert a char into ascii code. DataNucleus needs such function to converts chars to int values when performing queries converting chars to ints. DataNucleus defines a user defined function, DataNucleus_ASCII, which is automatically created on startup. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_ASCII;
CREATE FUNCTION NUCLEUS_ASCII(C CHAR(1)) RETURNS INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.ascii'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

String.matches(pattern): When pattern argument is a column, DataNucleus defines a function that allows Derby 10.1 to perform the matches function. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_MATCHES;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
CREATE FUNCTION NUCLEUS_MATCHES(TEXT VARCHAR(8000), PATTERN VARCHAR(8000)) RETURNS
INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.matches'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

Firebird

Firebird is supported as an RDBMS datastore by DataNucleus with the proviso that

- Auto-table creation is severely limited with Firebird. In Firebird, DDL statements are not auto-committed and are executed at the end of a transaction, after any DML statements. This makes "on the fly" table creation in the middle of a DML transaction not work. You must make sure that "autoStartMechanism" is NOT set to "SchemaTable" since this will use DML. You must also make sure that nobody else is connected to the database at the same time. Don't ask us why such limitations are in a RDBMS, but then it was you that chose to use it ;-)

To specify Firebird as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.firebirdsql.jdbc.FBDriver
datanucleus.ConnectionURL=jdbc:firebirdsql://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

SAPDB/MaxDB

To specify SAPDB/MaxDB as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=com.sap.dbtech.jdbc.DriverSapDB
datanucleus.ConnectionURL=jdbc:sapdb://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

JDBC Driver parameters

If you need to pass additional parameters to the JDBC driver you can append these to the end of the datanucleus.ConnectionURL. For example,

```
datanucleus.ConnectionURL=jdbc:mysql://localhost?useUnicode=true&characterEncoding=UTF-8
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

14.2.1 Java Types (Spatial)

Spatial Types Support

DataNucleus supports by default a large number of Java types. **DataNucleus Spatial** supports the storage and query of a number of different spatial data types, like points, polygons or lines. Spatial types like these are used to store geographic information like locations, rivers, cities, roads, etc. The datanucleus-spatial plugin allows using spatial and traditional types simultaneously in persistent objects making DataNucleus a single interface to read and manipulate any business data.

The table below shows the currently supported Spatial SCO Java types in DataNucleus

Java Type	Spec.	DFG?	Persistent?	Proxied?	PK?	Plugin
oracle.spatial.geome [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
com.vividsolutions.jts [1]	Extension					datanucleus-spatial
org.postgis.Geometr [1]	Extension					datanucleus-spatial
org.postgis.Geometr [1]	Extension					datanucleus-spatial
org.postgis.LinearRir [1]	Extension					datanucleus-spatial
org.postgis.LineStrin [1]	Extension					datanucleus-spatial

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Spec.	DFG?	Persistent?	Proxied?	PK?	Plugin
org.postgis.MultiLine [1]	Extension					datanucleus-spatial
org.postgis.MultiPoin [1]	Extension					datanucleus-spatial
org.postgis.MultiPoly [1]	Extension					datanucleus-spatial
org.postgis.Point [1]	Extension					datanucleus-spatial
org.postgis.Polygon [1]	Extension					datanucleus-spatial
org.postgis.PGbox2c [1]	Extension					datanucleus-spatial
org.postgis.PGbox3c [1]	Extension					datanucleus-spatial

- Dirty check mechanism is limited to immutable mode, it means, if you change a field of one of these spatial objects, you must reassign it to the owner object field to make sure changes are propagated to the database.

The implementation of these spatial types follows the OGC Simple Feature specification, but adds further types where the datastores support them.

Mapping Scenarios

DataNucleus supports different combinations of geometry libraries and spatially enabled databases. These combinations are called *mapping scenarios*. Each of these scenarios has a different set of advantages (and drawbacks), some have restrictions that apply. The table below tries to give as much information as possible about the different scenarios.

One such mapping scenario, is to use the Java geometry types from JTS (Java Topology Suite) and PostGIS as datastore. The short name for this mapping scenario is *jts2postgis*. The following table lists all supported mapping scenarios.

Geometry Libraries	MySQL [1]	Oracle [2] [4]	PostgreSQL with PostGIS [3] [4] [5]
Oracle's JGeometry	*jgeom2mysql*	*jgeom2oracle*	
Java Topology Suite (JTS)	*jts2mysql*	*jts2oracle*	*jts2postgis*
PostGIS JDBC Geometries	*pg2mysql*		*pg2postgis*

- **[1]** - MySQL doesn't support 3-dimensional geometries. Trying to persist them anyway results in **undefined behaviour**, there may be an exception thrown or the z-ordinate might just get stripped.

- **[2]** - Oracle Spatial supports additional data types like circles and curves that are not defined in the OGC SF specification. Any attempt to read or persist one of those data types, if you're not using *jgeom2oracle*, will result in failure!
- **[3]** - PostGIS added support for curves in version 1.2.0, but at the moment the JDBC driver doesn't support them yet. Any attempt to read curves geometries will result in failure, for every mapping scenario!
- **[4]** - Both PostGIS and Oracle have a system to add user data to specific points of a geometry. In PostGIS these types are called measure types and the z-coordinate of every 2d-point can be used to store arbitrary (numeric) data of double precision associated with that point. In Oracle this user data is called LRS. DataNucleus-Spatial tries to handle these types as gracefully as possible. But the recommendation is to not use them, unless you have a mapping scenario that is known to support them, i.e. *pg2postgis* for PostGIS and *jgeom2oracle* for Oracle.
- **[5]** - PostGIS supports two additional types called box2d and box3d, that are not defined in OGC SF. There are only mappings available for these types in *pg2postgis*, any attempt to read or persist one of those data types in another mapping scenario will result in failure!

Spatial types

This table lists all the spatial Java types that are currently supported. The JDBC type is the same for every Java type in a given database. It's `SDO_GEOMETRY` for Oracle, `OTHER` for PostGIS and `BINARY` for MySQL. When a type is supported by a database, the column type that is used for it, is listed after the icon. None of the types are proxied, this means that if you change an object field, you must reassign it to the owner object field to make sure changes are propagated to the database.

Geometry Library	Java Type	MySQL	Oracle	PostGIS
JGeometry	oracle.spatial.geometry.JGeometry	geometry	SDO_GEOMETRY	
JTS	com.vividsolutions.jts.geom.Geometry	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.GeometryCollection	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.LinearRing	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.LineString	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.MultiLineString	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.MultiPoint	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.MultiPolygon	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.Point	geometry	SDO_GEOMETRY	geometry
JTS	com.vividsolutions.jts.geom.Polygon	geometry	SDO_GEOMETRY	geometry

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Geometry Library	Java Type	MySQL	Oracle	PostGIS
PostGIS-JDBC	org.postgis.Geometry	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.GeometryCollection	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.LinearRing	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.LineString	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.MultiLineString	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.MultiPoint	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.MultiPolygon	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.Point	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.Polygon	✓ geometry	✗	✓ geometry
PostGIS-JDBC	org.postgis.PGbox2d	✗	✗	✓ box2d
PostGIS-JDBC	org.postgis.PGbox3d	✗	✗	✓ box3d

Metadata

DataNucleus-Spatial has defined some metadata extensions that can be used to give additional information about the geometry types in use. The position of these tags in the meta-data determines their scope. If you use them inside a <field>-tag the values are only used for that field specifically, if you use them inside the <package>-tag the values are in effect for all (geometry) fields of all classes inside that package, etc.

```
<package name="org.datanucleus.samples.jtsgeometry">
    <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/> [1]
    <extension vendor-name="datanucleus" key="spatial-srid" value="4326"/> [1]

    <class name="SampleGeometry" detachable="true">
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent">
            <extension vendor-name="datanucleus" key="mapping" value="no-userdata"/>
[2]
        </field>
    </class>

    <class name="SampleGeometryCollectionM" table="samplejtsgeometrycollectionm"
detachable="true">
        <extension vendor-name="datanucleus" key="postgis-hasMeasure" value="true"/>
[3]
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent"/>
    </class>

    <class name="SampleGeometryCollection3D" table="samplejtsgeometrycollection3d"
detachable="true">
        <extension vendor-name="datanucleus" key="spatial-srid" value="-1"/> [1]
        <extension vendor-name="datanucleus" key="spatial-dimension" value="3"/> [1]
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent"/>
    </class>
</package>
```

- **[1]** - The srid & dimension values are used in various places. One of them is schema creation, when using PostGIS, another is when you query the SpatialHelper.
- **[2]** - Every JTS geometry object can have a user data object attached to it. The default behaviour is to serialize that object and store it in a separate column in the database. If for some reason this isn't desired, the `mapping` extension can be used with value "no-mapping" and DataNucleus-Spatial will ignore the user data objects.
- **[3]** - If you want to use measure types in PostGIS you have to define that using the `postgis-hasMeasure` extension.

Querying Spatial types

DataNucleus-Spatial defines a set of functions that can be applied to spatial types in JDOQL queries. These functions follow the definitions in the OGC Simple Feature specification and are translated into appropriate SQL statements, provided the underlying database system implements the functions and the geometry object model accordingly. There are also some additional functions that are not defined OGC SF, most of them are database specific.

This set of more than eighty functions contains:

- Basic methods on geometry objects like `IsSimple()` and `Boundary()`.
- Methods for testing spatial relations between geometric objects like `Intersects()` and `Touches()`
- Methods that support spatial analysis like `Union()` and `Difference()`
- Methods to create geometries from WKB/WKT (Well Known Binary/Text) like `GeomFromText()` and `GeomFromWKB()`

For a compelete list of all supported functions and usage examples, please see JDOQL : Spatial Methods.

Dependencies

Depending on the mapping scenario you want to use, there is a different set of JARs that need to be in your classpath.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.2 Persistence Properties

RDBMS Persistence Properties

DataNucleus accepts a large number of persistence properties for use when defining either a PersistenceManagerFactory or an *EntityManagerFactory*. For RDBMS datastores there are several additional properties. These are listed here. Bear in mind that these only work with DataNucleus, and not with any other JDO/JPA implementation.

datanucleus.rdbms.datastoreAdapterClassName	
Description	This property allows you to supply the class name of the adapter to use for your datastore. The default is not to specify this property and DataNucleus will autodetect the datastore type and use its own internal datastore adapter classes. This allows you to override the default behaviour where there maybe is some issue with the default adapter class.
Range of Values	(valid class name on the CLASSPATH)

datanucleus.rdbms.statementBatchLimit	
Description	Maximum number of statements that can be batched. The default is 50 and also applies to delete of objects. Please refer to the Statement Batching guide
Range of Values	integer value (0 = no batching)

datanucleus.rdbms.checkExistTablesOrViews	
Description	Whether to check if the table/view exists. If false, it disables the automatic generation of tables that don't exist.
Range of Values	true | false

datanucleus.rdbms.initializeColumnInfo	
Description	Allows control over what column information is initialised when a table is loaded for the first time. By default info for all columns will be loaded. Unfortunately some RDBMS are particularly poor at returning this information so we allow reduced forms to just load the primary key column info, or not to load any.
Range of Values	**ALL** | PK | NONE

datanucleus.rdbms.classAdditionMaxRetries	
Description	The maximum number of retries when trying to find a class to persist or when validating a class.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.rdbms.classAdditionMaxRetries

Range of Values	3 | A positive integer

datanucleus.rdbms.constraintCreateMode

Description	How to determine the RDBMS constraints to be created. DataNucleus will automatically add foreign-keys/indices to handle all relationships, and will utilise the specified MetaData foreign-key information. JDO2 will only use the information in the MetaData file(s).
Range of Values	DataNucleus | JDO2

datanucleus.rdbms.uniqueConstraints.mapInverse

Description	Whether to add unique constraints to the element table for a map inverse field. Possible values are true or false.
Range of values	true | false

datanucleus.rdbms.discriminatorPerSubclassTable

Description	Property that controls if only the base class where the discriminator is defined will have a discriminator column
Range of values	false | true

datanucleus.rdbms.useUpdateLock

Description	Whether DataNucleus should use 'SELECT ... FOR UPDATE' on all fetch operations to prevent dirty writes. Only applies to read committed and read uncommitted transaction isolation level and doesn't apply to Optimistic Transactions. Please refer to the Transaction Types Guide for JDO and JPA
Range of Values	true | false

datanucleus.rdbms.stringDefaultLength

Description	The default (max) length to use for all strings that don't have their column length defined in MetaData.
Range of Values	256 (JDO) | 255 (JPA) | A valid length

datanucleus.rdbms.stringLengthExceededAction

Description	Defines what happens when persisting a String field and its length exceeds the length of the underlying datastore column. The default is to throw an Exception. The other option is to truncate the String to the length of the datastore column.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.rdbms.stringLengthExceededAction

Range of Values	EXCEPTION | TRUNCATE

datanucleus.rdbms.persistEmptyStringAsNull

Description	When persisting en empty string, should it be persisted as null in the datastore. This is to allow for datastores (Oracle) that dont differentiate between null and empty string. If it is set to false and the datastore doesnt differentiate then a special character will be saved when storing an empty string.
Range of Values	true | false

datanucleus.rdbms.sql.allowAllSQLStatements

Description	javax.jdo.query.SQL queries are allowed by JDO 2 only to run SELECT queries. This extension permits to bypass this limitation (so for example can execute stored procedures).
Range of Values	false | true

datanucleus.rdbms.query.fetchDirection

Description	The direction in which the query results will be navigated.
Range of Values	forward | reverse | unknown

datanucleus.rdbms.query.resultSetType

Description	Type of ResultSet to create. Note 1) Not all JDBC drivers accept all options. The values correspond directly to the ResultSet options. Note 2) Not all java.util.List operations are available for scrolling result sets. An Exception is raised when unsupported operations are invoked.
Range of Values	forward-only | scroll-sensitive | scroll-insensitive

datanucleus.rdbms.query.resultSetConcurrency

Description	Whether the ResultSet is readonly or can be updated. Not all JDBC drivers support all options. The values correspond directly to the ResultSet options.
Range of Values	read-only | updateable

datanucleus.rdbms.jdoql.joinType

Description	When running a JDOQL query, this defines what type of joins DataNucleus should try to use overriding the natural choice for the query. In some situations the user may know that it should use "INNER", or "LEFT OUTER" joins only.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.rdbms.jdoql.joinType

Range of Values	INNER | LEFT OUTER

datanucleus.rdbms.jdoql.existsIncludesConstraints

Description	When a JDOQL query has a "contains" clause this will be replaced by an EXISTS sub-query. This property defines whether all other clauses should be applied to this sub-query, instead of the parent query. In some cases this will lead to inefficient queries.
Range of Values	true | false

datanucleus.rdbms.oracleNlsSortOrder

Description	Sort order for Oracle String fields in queries (BINARY disables native language sorting)
Range of Values	LATIN | See Oracle documentation

datanucleus.rdbms.schemaTable.tableName

Description	Name of the table to use when using auto-start mechanism of "SchemaTable" Please refer to the RDBMS Auto-Start guide
Range of Values	NUCLEUS_TABLES | Valid table name

datanucleus.rdbms.connectionProviderName

Description	Name of the connection provider to use to allow failover Please refer to the Failover guide
Range of Values	**PriorityList** | Name of a provider

datanucleus.rdbms.connectionProviderFailOnError

Description	Whether to fail if an error occurs, or try to continue and log warnings
Range of Values	true | **false**

datanucleus.rdbms.dynamicSchemaUpdates

Description	Whether to allow dynamic updates to the schema. This means that upon each insert/update the types of objects will be tested and any previously unknown implementations of interfaces will be added to the existing schema.
Range of Values	true | **false**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.rdbms.omitDatabaseMetaDataGetColumns

Description	Whether to bypass all calls to DatabaseMetaData.getColumns(). This JDBC method is called to get schema information, but on some JDBC drivers (e.g Derby) it can take an inordinate amout of time. Setting this to true means that your datastore schema has to be correct and no checks will be performed.
Range of Values	true | **false**

datanucleus.rdbms.sqlTableNamingStrategy

Description	Name of the plugin to use for defining the names of the aliases of tables in SQL statements. This doesn't currently apply to JDOQL statements but will by version 1.2
Range of Values	**alpha-scheme** | t-scheme

datanucleus.rdbms.request.insert

Description	Name of plugin to use for inserts. See plugin point "org.datanucleus.store.rdbms.rdbms_request"
Range of Values	**default**

datanucleus.rdbms.request.update

Description	Name of plugin to use for updates. See plugin point "org.datanucleus.store.rdbms.rdbms_request"
Range of Values	**default**

datanucleus.rdbms.request.delete

Description	Name of plugin to use for deletes. See plugin point "org.datanucleus.store.rdbms.rdbms_request"
Range of Values	**default**

datanucleus.rdbms.request.fetch

Description	Name of plugin to use for fetches of fields. See plugin point "org.datanucleus.store.rdbms.rdbms_request"
Range of Values	**default**

datanucleus.rdbms.request.locate

Description	Name of plugin to use for locate of objects. See plugin point "org.datanucleus.store.rdbms.rdbms_request"

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.rdbms.request.locate

Range of Values	**default**

datanucleus.rdbms.tableColumnOrder

Description	How we should order the columns in a table. The default is to put the fields of the owning class first, followed by superclasses, then subclasses. An alternative is to start from the base superclass first, working down to the owner, then the subclasses
Range of Values	owner-first | superclass-first

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.3 Auto-Start

Automatic Startup

By default with JDO implementations when you open a PersistenceManagerFactory and obtain a PersistenceManager DataNucleus knows nothing about which classes are to be persisted to that datastore. JDO implementations only load the Meta-Data for any class when the class is first enlisted in a PersistenceManager operation. For example you call makePersistent on an object. The first time a particular class is encountered DataNucleus will dynamically load the Meta-Data for that class. This typically works well since in an application in a particular operation the PersistenceManagerFactory may well not encounter all classes that are persistable to that datastore. The reason for this dynamic loading is that JDO implementations can't be expected to scan through the whole Java CLASSPATH for classes that could be persisted there. That would be inefficient. There is an Auto-Start facility to alleviate this situation. RDBMS datastores allow a further auto-start mechanism option. This is described below.

SchemaTable

When using an RDBMS datastore the SchemaTable auto-start mechanism stores the list of classes (and their tables, types and version of DataNucleus) in a datastore table NUCLEUS_TABLES. This table is read at startup of DataNucleus, and provides DataNucleus with the necessary knowledge it needs to continue persisting these classes. This table is continuously updated during a session of a DataNucleus-enabled application. This is the default setting for the auto-start mechanism so, unless you change the datanucleus.autoStartMechanism property, you will have a table NUCLEUS_TABLES created in your datastore schema.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from NUCLEUS_TABLES and finds that a table/class required by the NUCLEUS_TABLES data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property datanucleus.autoStartMechanismMode defines the behaviour of DataNucleus for this situation.

- Checked will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).
- Quiet (the default) will simply remove the entry from NUCLEUS_TABLES and continue without exception.
- Ignored will simply continue without doing anything.

The default database schema used the *SchemaTable* is described below:

```
TABLE : NUCLEUS_TABLES
(
    COLUMN : CLASS_NAME VARCHAR(128) PRIMARY KEY,  -- Fully qualified persistent
Class name
    COLUMN : TABLE_NAME VARCHAR(128),              -- Table name
    COLUMN : TYPE VARCHAR(4),                      -- FCO | SCO
    COLUMN : OWNER VARCHAR(2),                     -- 1 | 0
    COLUMN : VERSION VARCHAR(20),                  -- DataNucleus version
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    COLUMN : INTERFACE_NAME VARCHAR(255)           -- Fully qualified persistent
Class type
                                                   -- of the persistent Interface
implemented
)
```

If you want to change the table name (from NUCLEUS_TABLES) you can set the persistence property *datanucleus.rdbms.schemaTable.tableName*

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

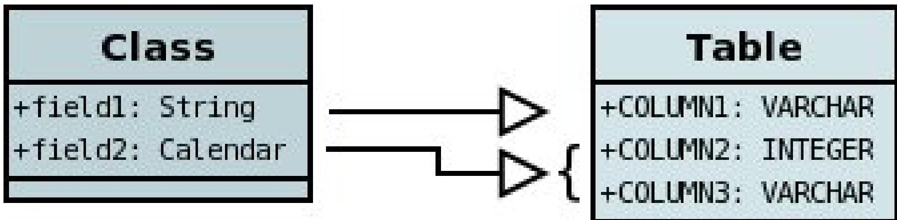
14.2.4 Datastore Types

RDBMS : Datastore Types

As we saw in the Types Guide DataNucleus supports the persistence of a large range of Java field types. With RDBMS datastores, we have the notion of tables/columns in the datastore and so each Java type is mapped across to a column or a set of columns in a table. It is important to understand this mapping when mapping to an existing schema for example. In RDBMS datastores a java type is stored using JDBC types. DataNucleus supports the use of the vast majority of the available JDBC types.

JDBC types used when persisting Java types

When persisting a Java type in general it is persisted into a single column. For example a String will be persisted into a VARCHAR column by default. Some types (e.g Color) have more information to store than we can conveniently persist into a single column and so use multiple columns. Other types (e.g Collection) store their information in other ways, such as foreign keys.



This table shows the Java types we saw earlier and whether they can be queried using JDOQL queries, and what JDBC types can be used to store them in your RDBMS datastore. Not all RDBMS datastores support all of these options. While DataNucleus always tries to provide a complete list sometimes this is impossible due to limitations in the underlying JDBC driver

Java Type	Number Columns	Queryable	JDBC Type(s)
boolean	1	✓	BIT, CHAR ('Y','N'), BOOLEAN, TINYINT, SMALLINT, NUMERIC
byte	1	✓	TINYINT, SMALLINT, NUMERIC
char	1	✓	CHAR, INTEGER, NUMERIC
double	1	✓	DOUBLE, DECIMAL, FLOAT
float	1	✓	FLOAT, REAL, DOUBLE, DECIMAL
int	1	✓	INTEGER, BIGINT, NUMERIC
long	1	✓	BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Number Columns	Queryable	JDBC Type(s)
short	1		SMALLINT, INTEGER, NUMERIC
boolean[]	1	[6]	LONGVARBINARY, BLOB
byte[]	1	[6]	LONGVARBINARY, BLOB
char[]	1	[6]	LONGVARBINARY, BLOB
double[]	1	[6]	LONGVARBINARY, BLOB
float[]	1	[6]	LONGVARBINARY, BLOB
int[]	1	[6]	LONGVARBINARY, BLOB
long[]	1	[6]	LONGVARBINARY, BLOB
short[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Boolean	1		BIT, CHAR('Y','N'), BOOLEAN, TINYINT, SMALLINT
java.lang.Byte	1		TINYINT, SMALLINT, NUMERIC
java.lang.Character	1		CHAR, INTEGER, NUMERIC
java.lang.Double	1		DOUBLE, DECIMAL, FLOAT
java.lang.Float	1		FLOAT, REAL, DOUBLE, DECIMAL
java.lang.Integer	1		INTEGER, BIGINT, NUMERIC
java.lang.Long	1		BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER
java.lang.Short	1		SMALLINT, INTEGER, NUMERIC
java.lang.Boolean[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Byte[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Character[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Double[]	1	[6]	LONGVARBINARY, BLOB

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Number Columns	Queryable	JDBC Type(s)
java.lang.Float[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Integer[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Long[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Short[]	1	[6]	LONGVARBINARY, BLOB
java.lang.Number	1		
java.lang.Object	1		LONGVARBINARY, BLOB
java.lang.String [9]	1		VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.lang.StringBuffer [9]	1		VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.lang.String[]	1	[6]	LONGVARBINARY, BLOB
java.math.BigDecimal	1		DECIMAL, NUMERIC
java.math.BigInteger	1		NUMERIC, DECIMAL
java.math.BigDecimal[]	1	[6]	LONGVARBINARY, BLOB
java.math.BigInteger[]	1	[6]	LONGVARBINARY, BLOB
java.sql.Date	1		DATE, TIMESTAMP
java.sql.Time	1		TIME, TIMESTAMP
java.sql.Timestamp	1		TIMESTAMP
java.util.ArrayList	0		
java.util.BitSet	0		LONGVARBINARY, BLOB
java.util.Calendar [3]	1 or 2		INTEGER, VARCHAR, CHAR
java.util.Collection	0		

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Number Columns	Queryable	JDBC Type(s)
java.util.Currency	1		VARCHAR, CHAR
java.util.Date	1		TIMESTAMP, DATE, CHAR, BIGINT
java.util.Date[]	1	[6]	LONGVARBINARY, BLOB
java.util.GregorianCalendar [2]	1 or 2		INTEGER, VARCHAR, CHAR
java.util.HashMap	0		
java.util.HashSet	0		
java.util.Hashtable	0		
java.util.LinkedHashMap	0		
java.util.LinkedHashSet	0		
java.util.LinkedList	0		
java.util.List	0		
java.util.Locale [9]	1		VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.util.Locale[]	1	[6]	LONGVARBINARY, BLOB
java.util.Map	0		
java.util.Properties	0		
java.util.PriorityQueue	0		
java.util.Queue	0		
java.util.Set	0		
java.util.SortedMap	0		
java.util.SortedSet	0		
java.util.Stack	0		

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Number Columns	Queryable	JDBC Type(s)
java.util.TimeZone [9]	1	✓	VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.util.TreeMap	0	✓	
java.util.TreeSet	0	✓	
java.util.UUID [9]	1	✓	VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.util.Vector	0	✓	
java.awt.Color [1]	4	✗	INTEGER
java.awt.Point [2]	2	✗	INTEGER
java.awt.image.BufferedImage [5]	1	✗	LONGVARBINARY, BLOB
java.net.URI [9]	1	✓	VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.net.URL [9]	1	✓	VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]
java.io.Serializable	1	✗	LONGVARBINARY, BLOB
javax.jdo.spi.PersistenceCapable	1	✓	[embedded]
javax.jdo.spi.PersistenceCapable[]	1	✓ [6]	
java.lang.Enum [4]	1	✓	LONGVARBINARY, BLOB, VARCHAR, INTEGER

- [1] - java.awt.Color - stored in 4 columns (red, green, blue, alpha). ColorSpace is not persisted.
- [2] - java.awt.Point - stored in 2 columns (x and y).
- [3] - java.util.Calendar - stored in 2 columns (milliseconds and timezone).
- [4] - java.lang.Enum - by default, serialized into one column. Storing in VARCHAR or INTEGER data types is available via the "Java5" plugin. Queryable if stored in VARCHAR or INTEGER columns.
- [5] - java.awt.image.BufferedImage is stored using JPG image format

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- [6] - Array types are queryable if not serialised, but stored to many rows
- [7] - DATALINK JDBC type supported on DB2 only. Uses the SQL function DLURLCOMPLETEONLY to fetch from the datastore. You can override this using the select-function extension. See the JDO MetaData reference.
- [8] - UNIQUEIDENTIFIER JDBC type supported on MSSQL only.
- [9] - Oracle treats an empty string as the same as NULL. To workaround this limitation DataNucleus replaces the empty string with the character \u0001.
- [10] - XMLTYPE JDBC type supported on Oracle only, and is included in the "datanucleus-xmltypeoracle" plugin.

If you need to extend the provided DataNucleus capabilities in terms of its datastore types support you can utilise a plugin point.

Supported JDBC types

DataNucleus provides support for the majority of the JDBC types. The support is shown below.

JDBC Type	Supported	Restrictions
ARRAY	✗	
BIGINT	✓	
BINARY	✓	Only for spatial types on MySQL
BIT	✓	
BLOB	✓	
BOOLEAN	✓	
CHAR	✓	
CLOB	✓	
DATALINK	✓	Only on DB2
DATE	✓	
DECIMAL	✓	
DISTINCT	✗	
DOUBLE	✓	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

JDBC Type	Supported	Restrictions
FLOAT	Yes	
INTEGER	Yes	
JAVA_OBJECT	No	
LONGVARBINARY	Yes	
LONGVARCHAR	Yes	
NULL	No	
NUMERIC	Yes	
OTHER	Yes	Only for spatial types on PostgreSQL with PostGIS extension
REAL	Yes	
REF	No	
SMALLINT	Yes	
STRUCT	Yes	Only for spatial types on Oracle
TIME	Yes	
TIMESTAMP	Yes	
TINYINT	Yes	
VARBINARY	Yes	
VARCHAR	Yes	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.5 Datasource

RDBMS Data Sources

DataNucleus requires a data source that represents the datastore in use. This is often just a URL defining the location of the datastore, but (in the case of RDBMS) there are in fact several ways of specifying this data source depending on the environment in which you are running.

- Stand Alone Environment
- Nonmanaged Context - Java Client
- Managed Context - Servlet
- Managed Context - J2EE

Stand-Alone Environment : Connection Pooling

When running a stand-alone application (not within a J2EE environment), you would typically specify properties for your PersistenceManagerFactory as in the following example

```
datanucleus.ConnectionDriverName=com.mysql.jdbc.Driver
datanucleus.ConnectionURL=jdbc:mysql://localhost/myDB
datanucleus.ConnectionUserName=...
datanucleus.ConnectionPassword=...
```

So in this case we have a MySQL datastore, with the database called "myDB". This is perhaps the most common specification method, but *does not use connection pool*. You can specify connection pooling with this method - see the RDBMS Connection Pooling Guide.

See also :

- LongPlay Web Tutorial

Java Client Environment : Nonmanaged Context

DataNucleus permits you to take advantage of using database connection pooling that is available on an application server. The application server could be a full J2EE server (e.g WebLogic) or could equally be a servlet engine (e.g Tomcat, Jetty). Here we are in a non-managed context, and we use the following properties when creating our PersistenceManagerFactory, and refer to the JNDI data source of the server.

If the data source is avaiable in WebLogic, the simplest way of using a data source outside the application server is as follows.

```
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,"t3://localhost:7001");

Context ctx = new InitialContext(ht);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                    "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactory",ds);

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

If the data source is avaiable in Websphere, the simplest way of using a data source outside the application server is as follows.

```
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"com.ibm.websphere.naming.WsnInitialContextFactory");
ht.put(Context.PROVIDER_URL,"iiop://server:orb port");

Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                    "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactory",ds);

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

Servlet Environment : Managed Context

As an example of setting up such a JNDI data source for Tomcat 5.0, here we would add the following file to $TOMCAT/conf/Catalina/localhost/ as "datanucleus.xml"

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/home/datanucleus/" path="/datanucleus">
    <Resource name="jdbc/datanucleus" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/datanucleus">
        <parameter>
            <name>maxWait</name>
            <value>5000</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>20</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>2</value>
        </parameter>

        <parameter>
            <name>url</name>
<value>jdbc:mysql://127.0.0.1:3306/datanucleus?autoReconnect=true</value>
        </parameter>
        <parameter>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <name>driverClassName</name>
            <value>com.mysql.jdbc.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>mysql</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value></value>
        </parameter>
    </ResourceParams>
</Context>
```

With this Tomcat JNDI data source we would then specify the PMF ConnectionFactoryName as java:comp/env/jdbc/datanucleus.

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                    "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactoryName","java:comp/env/jdbc/datanucleus");

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

See also :

- LongPlay Web Tutorial

J2EE Environment : Managed Context

As in the above example, we can also run in a managed context, in a J2EE/Servlet environment, and here we would make a minor change to the specification of the JNDI data source depending on the application server or the scope of the jndi: global or component.

Using JNDI deployed in global environment:

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                    "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactoryName","jdbc/datanucleus");

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

Using JNDI deployed in component environment:

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                    "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactoryName","java:comp/env/jdbc/datanucleus");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

See also :

- J2EE Tutorial for JDO

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.6 Connection Pooling

RDBMS Connection Pools

When you create a PersistenceManagerFactory you define the connection URL, driver name, and the username/password to use. This works perfectly well but does not "pool" the connections so that they are efficiently opened/closed when needed to utilise datastore resources in an optimum way. DataNucleus allows you to utilise a connection pool to efficiently manage the connections to the datastore. You can do this using one of the following ways currently.

- DataNucleus DBCP plugin
- DataNucleus C3P0 plugin
- DataNucleus Proxool plugin
- Using JNDI, and lookup a connection DataSource.
- Manually creating a DataSource for a 3rd party software package
- Custom DataSource Pooling Plugins using the DataNucleusDataSourceFactory interface

Lookup a DataSource using JNDI

DataNucleus allows you to use connection pools (java.sql.DataSource) bound to a javax.naming.InitialContext with a JNDI name. You first need to create the DataSource in the container (application server/web server), and secondly you define the **datanucleus.ConnectionFactoryName** property with the DataSource JDNI name.

The following example uses a properties file that is loaded before creating the PersistenceManagerFactory. The PersistenceManagerFactory is created using the JDOHelper.

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
datanucleus.ConnectionFactoryName=YOUR_DATASOURCE_JNDI_NAME
```

```
Properties properties = new Properties();

// the properties file is in your classpath
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("/yourpath/yourfile.properties");
```

Please read more about this in the Data Source Guide.

Manually create a DataSource ConnectionFactory

We could have used the DataNucleus plugin which internally creates a DataSource ConnectionFactory, however we can also do this manually if we so wish. Let's demonstrate how to do this with one of the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

most used pools Jakarta DBCP

With DBCP you need to generate a javax.sql.DataSource, which you will then pass to DataNucleus. You do this as follows

```
// Load the JDBC driver
Class.forName(dbDriver);

// Create the actual pool of connections
ObjectPool connectionPool = new GenericObjectPool(null);

// Create the factory to be used by the pool to create the connections
ConnectionFactory connectionFactory = new DriverManagerConnectionFactory(dbURL,
dbUser, dbPassword);

// Create a factory for caching the PreparedStatements
KeyedObjectPoolFactory kpf = new StackKeyedObjectPoolFactory(null, 20);

// Wrap the connections with pooled variants
PoolableConnectionFactory pcf = new PoolableConnectionFactory(connectionFactory,
connectionPool, kpf,
                                                              null, false, true);

// Create the datasource
DataSource ds = new PoolingDataSource(connectionPool);

// Create our PMF for using DataNucleus
PersistenceManagerFactory pmf = new
org.datanucleus.jdo.JDOPersistenceManagerFactory();
pmf.setConnectionDriverName(dbDriver);
pmf.setConnectionURL(dbURL);
pmf.setConnectionFactory(ds);

// Set any other properties on the PMF.
```

As you see use the PMF constructor to create our factory. The reason is that the JDOHelper alternative doesn't allow us to specify the data source to be used. Also we haven't passed the dbUser and dbPassword to the PMF since we no longer need to specify them - they are defined for the pool so we let it do the work. As you also see, we use the pmf.setConnectionFactory() method to assign the data source to the PMF. Thereafter we can sit back and enjoy the performance benefits. Please refer to the documentation for DBCP for details of its configurability (you will need commons-dbcp, commons-pool, and commons-collections in your CLASSPATH to use this above example).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.6.1 C3P0

C3P0 Connection Pools

When you create a PersistenceManagerFactory/*EntityManagerFactory* you define the connection URL, driver name, and the username/password to use. This works perfectly well but does not "pool" the connections so that they are efficiently opened/closed when needed to utilise datastore resources in an optimum way. DataNucleus allows you to utilise a connection pool using C3P0 to efficiently manage the connections to the datastore.

C3P0 is a third-party library providing connection pooling. This is accessed by specifying the persistence property datanucleus.connectionPoolingType. To utilise C3P0-based connection pooling we do this

```
// Specify our persistence properties used for creating our PMF/EMF
Properties props = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
properties.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
properties.setProperty("datanucleus.ConnectionUserName","login");
properties.setProperty("datanucleus.ConnectionPassword","password");
properties.setProperty("datanucleus.connectionPoolingType", "C3P0");
```

So the PMF/*EMF* will use connection pooling using C3P0. To do this you will need the *datanucleus-connectionpool* plugin to be in the CLASSPATH, along with the C3P0 JAR. If you want to configure C3P0 further you can include a "c3p0.properties" in your CLASSPATH - see the C3P0 documentation for details.

You can also specify persistence properties to control the actual pooling. The currently supported properties for C3P0 are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.minPoolSize=3
datanucleus.connectionPool.initialPoolSize=3

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=20
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.6.2 DBCP

Apache DBCP Connection Pools

When you create a PersistenceManagerFactory/*EntityManagerFactory* you define the connection URL, driver name, and the username/password to use. This works perfectly well but does not "pool" the connections so that they are efficiently opened/closed when needed to utilise datastore resources in an optimum way. DataNucleus allows you to utilise a connection pool using Apache DBCP to efficiently manage the connections to the datastore.

DBCP is a third-party library providing connection pooling. This is accessed by specifying the persistence property datanucleus.connectionPoolingType. To utilise DBCP-based connection pooling we do this

```
// Specify our persistence properties used for creating our PMF/EMF
Properties props = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
properties.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
properties.setProperty("datanucleus.ConnectionUserName","login");
properties.setProperty("datanucleus.ConnectionPassword","password");
properties.setProperty("datanucleus.connectionPoolingType", "DBCP");
```

So the PMF/*EMF* will use connection pooling using DBCP. To do this you will need the *datanucleus-connectionpool* plugin to be in the CLASSPATH, along with the commons-dbcp, commons-pool and commons-collections JARs.

You can also specify persistence properties to control the actual pooling. The currently supported properties for DBCP are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxIdle=10
datanucleus.connectionPool.minIdle=3
datanucleus.connectionPool.maxActive=5
datanucleus.connectionPool.maxWait=60

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=20

datanucleus.connectionPool.testSQL=SELECT 1

datanucleus.connectionPool.timeBetweenEvictionRunsMillis=2400000
datanucleus.connectionPool.minEvictableIdleTimeMillis=18000000
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.6.3 Proxool

Proxool Connection Pool

When you create a PersistenceManagerFactory/*EntityManagerFactory* you define the connection URL, driver name, and the username/password to use. This works perfectly well but does not "pool" the connections so that they are efficiently opened/closed when needed to utilise datastore resources in an optimum way. DataNucleus allows you to utilise a connection pool using Proxool to efficiently manage the connections to the datastore.

Proxool is a third-party library providing connection pooling. This is accessed by specifying the persistence property datanucleus.connectionPoolingType. To utilise Proxool-based connection pooling we do this

```
// Specify our persistence properties used for creating our PMF/EMF
Properties props = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                "org.datanucleus.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
properties.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
properties.setProperty("datanucleus.ConnectionUserName","login");
properties.setProperty("datanucleus.ConnectionPassword","password");
properties.setProperty("datanucleus.connectionPoolingType", "Proxool");
```

So the PMF/*EMF* will use connection pooling using Proxool. To do this you will need the *datanucleus-connectionpool* plugin to be in the CLASSPATH, along with the proxool and commons-logging JARs.

You can also specify persistence properties to control the actual pooling. The currently supported properties for Proxool are shown below

```
datanucleus.connectionPool.maxConnections=10

datanucleus.connectionPool.testSQL=SELECT 1
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7 Queries

RDBMS Queries

Using an RDBMS datastore DataNucleus allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- SQL - language found on alomst all RDBMS.
- JPQL - language defined in the JPA1 specification for JPA persistence which closely mirrors SQL.

When using queries with RDBMS there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Query Timeouts

DataNucleus provides a useful extension allowing control over the timeout of the query. So, for example, if you have a query that can cause problems in terms of the time taken, you can set a timeout on the query to retain the usability of your application.

With JDO2 you can set this on a per-Query basis by specifying the query "extension" **datanucleus.query.timeout** setting it to the value (in millisecs) that you want the timeout to be. With JDO/JPA you can apply it to all queries via a persistence property **datanucleus.query.timeout**

Result Set : Type

java.sql.ResultSet defines three possible result set types.

- *forward-only* : the result set is navegable forwards only
- *scroll-sensitive* : the result set is scrollable in both directions and is sensitive to changes in the datastore
- *scroll-insensitive* : the result set is scrollable in both directions and is insensitive to changes in the datastore

DataNucleus allows specification of this type as a query extension **datanucleus.rdbms.query.resultSetType**.

To do this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.rdbms.query.resultSetType", "scroll-insensitive");
```

To do this on a per query basis for JPA1 you would do

```

```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
query.setHint("datanucleus.rdbms.query.resultSetType", "scroll-insensitive");
```

The default is *forward-only*. The benefit of the other two is that the result set will be scrollable and hence objects will only be read in to memory when accessed. So if you have a large result set you should set this to one of the scrollable values.

Result Set : Caching of Results

When using a "scrollable" result set (see above for **datanucleus.rdbms.query.resultSetType**) by default the query result will cache the rows that have been read. You can control this caching to optimise it for your memory requirements. You can set the query extension **datanucleus.query.resultCacheType** and it has the following possible values

- *weak* : use a weak hashmap for caching (default)
- *soft* : use a soft reference map for caching
- *hard* : use a HashMap for caching (objects not garbage collected)
- *none* : no caching (hence uses least memory)

To set this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.query.resultCacheType", "weak");
```

To do this on a per query basis for JPA1 you would do

```
query.setHint("datanucleus.query.resultCacheType", "weak");
```

Large Result Sets : Size

If you have a large result set you clearly don't want to instantiate all objects since this would hit the memory footprint of your application. To get the number of results many JDBC drivers will load all rows of the result set. This is to be avoided so DataNucleus provides control over the mechanism for getting the size of results. The persistence property **datanucleus.query.resultSizeMethod** has a default of *last* (which means navigate to the last object - hence hitting the JDBC driver problem). If you set this to *count* then it will use a simple "count()" query to get the size.

To do this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.query.resultSizeMethod", "count");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

To do this on a per query basis for JPA1 you would do

```
query.setHint("datanucleus.query.resultSizeMethod", "count");
```

Large Result Sets : Loading Results at Commit()

When a transaction is committed by default all remaining results for a query are loaded so that the query is usable thereafter. With a large result set you clearly don't want this to happen. So in this case you should set the extension **datanucleus.query.loadResultsAtCommit** to false.

To do this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.query.loadResultsAtCommit", "false");
```

To do this on a per query basis for JPA1 you would do

```
query.setHint("datanucleus.query.loadResultsAtCommit", "false");
```

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPA1 you would do

```
query.setHint("datanucleus.query.flushBeforeExecution","true");
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Result Set : Control

DataNucleus provides a useful extension allowing control over the ResultSet's that are created by queries. You have at your convenience some properties that give you the power to control whether the result set is read only, whether it can be read forward only, the direction of fetching etc.

To do this on a per query basis for JDO2 you would do

```
query.addExtension("datanucleus.rdbms.query.fetchDirection", "forward");
query.addExtension("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

To do this on a per query basis for JPA1 you would do

```
query.setHint("datanucleus.rdbms.query.fetchDirection", "forward");
query.setHint("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

Alternatively you can specify these as persistence properties so that they apply to all queries for that PMF/EMF. Again, the properties are

- **datanucleus.rdbms.query.fetchDirection** - controls the direction that the ResultSet is navigated. By default this is forwards only. Use this property to change that.
- **datanucleus.rdbms.query.resultSetConcurrency** - controls whether the ResultSet is read only or updateable.

Bear in mind that not all RDBMS support all of the possible values for these options. That said, they do add a degree of control that is often useful.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7.1 JDOQL

RDBMS JDOQL Queries

As shown in JDOQL Reference DataNucleus supports queries using the JDOQL query language, using a Java-like syntax. When using JDOQL under RDBMS there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Control over Joins

JDOQL queries are converted into SQL and use a mix of INNER and LEFT OUTER joins. In some situations the chosen join may not be optimal and you may know that using only INNER joins would be more efficient. If you have this situation you can use a DataNucleus extension to enable this behaviour. **datanucleus.rdbms.jdoql.joinType**, has valid values of "INNER" and "LEFT OUTER" and DataNucleus will attempt to make joins of that type when compiling the query. This can be specified as either a JDOQL extension (apply to this query only), or as a PMF property (apply to all queries).

Control over locking of fetched objects

DataNucleus allows control over whether objects found by a fetch (JDOQL query) are locked during that transaction so that other transactions can't update them in the meantime. To do this you would do

```
Query q = pm.newQuery(...);
((org.datanucleus.jdo.JDOQuery)q).setSerializeRead(true);
```

You can also specify this for all queries for all PMs using a PMF property **datanucleus.SerializeRead**. In addition you can perform this on a per-transaction basis by doing

```
((org.datanucleus.jdo.JDOTransaction)pm.currentTransaction()).serializeRead(true);
```

Use of contains() and variables

When using the method *contains* on a collection (or *containsKey*, *containsValue* on a map) this will add an "EXISTS" subquery to the executed SQL. This can have an impact on where the query should apply other conditions for the variable in order to get correct syntax. Let's take an example. We have classes A and B, with A having a Set of Bs (using ForeignKey).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
SELECT FROM org.datanucleus.samples.A
WHERE type == 'Technology' && elements.contains(b1) && b1.name == 'Jones'
VARIABLES org.datanucleus.samples.B b1
```

By default this will create a statement like

```
SELECT `THIS`.`ID`
FROM `A` `THIS`
WHERE EXISTS (
    SELECT 1 FROM `B` `THIS_ELEMENTS_B1`
    WHERE `THIS_ELEMENTS_B1`.`A_ID` = `THIS`.`ID`
    AND `THIS`.`TYPE` = 'Technology'
    AND `THIS_ELEMENTS_B1`.`NAME` = 'Jones'
)
```

So, as you can see, the restriction on "A.type" is applied to the EXISTS subquery. This is not as efficient as it could be. We would like the "type == 'Technology'" to be applied to the main query, and the "b1.name == 'Jones'" to be applied to the subquery. To do this we need to change the query

```
SELECT FROM org.datanucleus.samples.A
WHERE type == 'Technology' && (elements.contains(b1) && b1.name == 'Jones')
VARIABLES org.datanucleus.samples.B b1
```

So we used **parentheses (brackets)** to reinforce that the variable constraints should be treated together (and hence in the same subquery), and we now use an extension **datanucleus.rdbms.jdoql.existsIncludesConstraints"**, set to "false". This can be specified as either a JDOQL extension (apply to this query only), or as a PMF property (apply to all queries). This means that the "type == 'Technology'" is applied to the main query and **not** to the EXISTS subquery.

```
SELECT `THIS`.`ID`
FROM `A` `THIS`
WHERE `THIS`.`TYPE` = 'Technology'
AND (EXISTS (
    SELECT 1 FROM `B` `THIS_ELEMENTS_B1`
    WHERE `THIS_ELEMENTS_B1`.`A_ID` = `THIS`.`ID`
    AND `THIS_ELEMENTS_B1`.`PROPERTY_NAME` = 'Jones'
)
```

A further extension is useful where, in some situation, when having *contains(this.field)* or *contains(this)* and this doesnt use an EXISTS statement. To force it to use EXISTS you should specify the query extension **datanucleus.rdbms.query.containsUsesExistsAlways** as "true". The query will then use EXISTS for that contains clause.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

14.2.7.2 JPQL

RDBMS JPQL Queries

As shown in JPQL Reference DataNucleus supports queries using the JPQL query language, using a Java-like syntax. Here we provide details of the implementation of JPQL for RDBMS datastores.

Control over locking of fetched objects

DataNucleus allows control over whether objects found by a fetch (JPQL query) are locked during that transaction so that other transactions can't update them in the meantime. You can use the DataNucleus extension **datanucleus.rdbms.query.useUpdateLock**, set to "true", and this will append "FOR UPDATE" on the end of the SELECT. This can be specified as either a JPQL hint (apply to this query only), or as an EMF property (apply to all queries).

You can also specify this for all queries for all PMs/EMs using a persistence property **datanucleus.rdbms.useUpdateLock**.

SQL Generation

With a JPQL query running on an RDBMS the query is compiled into SQL. Here we give a few examples of what SQL is generated. You can of course try this for yourself observing the content of the DataNucleus log.

In JPQL you specify a candidate class and its alias (identifier). In addition you can specify joins with their respective alias. The DataNucleus implementation of JPQL will preserve these aliases in the generated SQL.

```
JPQL:
SELECT Object(P) FROM mydomain.Person P INNER JOIN P.bestFriend AS B

SQL:
SELECT P.ID
FROM PERSON P INNER JOIN PERSON B ON B.ID = P.BESTFRIEND_ID
```

With the JPQL MEMBER OF syntax this is typically converted into an EXISTS query.

```
JPQL:
SELECT DISTINCT Object(p) FROM mydomain.Person p WHERE :param MEMBER OF p.friends

SQL:
SELECT DISTINCT P.ID FROM PERSON P
WHERE EXISTS (
    SELECT 1 FROM PERSON_FRIENDS P_FRIENDS, PERSON P_FRIENDS_1
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
WHERE P_FRIENDS.PERSON_ID = P.ID
AND P_FRIENDS_1.GLOBAL_ID = P_FRIENDS.FRIEND_ID
AND 101 = P_FRIENDS_1.ID)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7.3 JDOQL : Methods

JDOQL : Methods

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. In addition, DataNucleus adds on many extensions to the JDO specifications providing all of the commonly required methods. Below is a list of what is required by JDOQL in JDO 1.0 and JDO 2.0, and what DataNucleus RDBMS supports

Java Type	Method	Description	Specification	DataNucleus support
String	startsWith(String)	Returns if the string starts with the passed string	JDO 1.0, JDO 2.0	
String	endsWith(String)	Returns if the string ends with the passed string	JDO 1.0, JDO 2.0	
String	indexOf(String)	Returns the first position of the passed string	JDO 2.0	
String	indexOf(String,int)	Returns the position of the passed string, after the passed position	JDO 2.0	
String	substring(int)	Returns the substring starting from the passed position	JDO 2.0	
String	substring(int,int)	Returns the substring between the passed positions	JDO 2.0	
String	toLowerCase()	Returns the string in lowercase	JDO 2.0	
String	toUpperCase()	Retuns the string in UPPERCASE	JDO 2.0	
String	matches(String pattern)	Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method.	JDO 2.0	
String	like(String pattern)	Returns whether string matches the passed expression. The pattern argument is database specific.	Extension	
String	charAt(int)	Returns the character at the passed position	Extension	
String	startsWith(String,int)	Returns if the string starts with the passed string, from the passed position	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
String	length()	Returns the length of the string	Extension	
String	equals(String)	Returns if the string is equals to the passed string	Extension	
String	trim()	Returns a trimmed version of the string	Extension	
String	trimLeft()	Returns a trimmed version of the string (trimmed for leading spaces)	Extension	
String	trimRight()	Returns a trimmed version of the string (trimmed for trailing spaces)	Extension	
String	translate(to,from) [2]	Translates the "from" characters into "to"	Extension	
Collection	isEmpty()	Returns whether the collection is empty	JDO 1.0, JDO 2.0	
Collection	contains(value)	Returns whether the collection contains the passed element	JDO 1.0, JDO 2.0	
Collection	size()	Returns the number of elements in the collection	JDO 2.0	
Map	isEmpty()	Returns whether the map is empty	JDO 1.0, JDO 2.0	
Map	contains(value)	Returns whether the map contains the passed value	Extension	
Map	containsKey(key)	Returns whether the map contains the passed key	JDO 2.0	
Map	containsValue(value)	Returns whether the map contains the passed value	JDO 2.0	
Map	containsEntry(key,value)	Returns whether the map contains the passed entry	Extension	
Map	get(key)	Returns the value from the map with the passed key	JDO 2.0	
Map	size()	Returns the number of entries in the map	JDO 2.0	
Math	abs(number)	Returns the absolute value of the passed number	JDO 2.0	
Math	sqrt(number)	Returns the square root of the passed number	JDO 2.0	
Math	cos(number)	Returns the cosine of the passed number	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
Math	sin(number)	Returns the absolute value of the passed number	Extension	
Math	tan(number)	Returns the tangent of the passed number	Extension	
Math	acos(number)	Returns the arc cosine of the passed number	Extension	
Math	asin(number)	Returns the arc sine of the passed number	Extension	
Math	atan(number)	Returns the arc tangent of the passed number	Extension	
Math	exp(number)	Returns the exponent of the passed number	Extension	
Math	log(number)	Returns the log(base e) of the passed number	Extension	
Math	floor(number)	Returns the floor of the passed number	Extension	
Math	ceil(number)	Returns the ceiling of the passed number	Extension	
Date	getDay()	Returns the day (of the month) for the date	Extension	
Date	getMonth()	Returns the month for the date	Extension	
Date	getYear()	Returns the year for the date	Extension	
Time	getHour()	Returns the hour for the time	Extension	
Time	getMinute()	Returns the minute for the time	Extension	
Time	getSecond()	Returns the second for the time	Extension	
JDOHelper	getObjectId(object)	Returns the object identity of the passed persistent object	JDO 2.0	
Analysis	rollup({object})	Perform a rollup operation over the results.	Extension	
Analysis	cube({object})	Perform a cube operation over the results.	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
{}	length	Returns the length of an array	Extension	
{}	contains(object)	Returns true if the array contains the object	Extension	
Enum	startsWith(String) [1]	Returns if the string starts with the passed string	Extension	
Enum	endsWith(String) [1]	Returns if the string ends with the passed string	Extension	
Enum	indexOf(String) [1]	Returns the first position of the passed string	Extension	
Enum	indexOf(String,int) [1]	Returns the position of the passed string, after the passed position	Extension	
Enum	substring(int) [1]	Returns the substring starting from the passed position	Extension	
Enum	substring(int,int) [1]	Returns the substring between the passed positions	Extension	
Enum	toLowerCase() [1]	Returns the string in lowercase	Extension	
Enum	toUpperCase() [1]	Retuns the string in UPPERCASE	Extension	
Enum	matches(String pattern) [1]	Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method.	Extension	
Enum	like(String pattern) [1]	Returns whether string matches the passed expression. The pattern argument is database specific.	Extension	
Enum	charAt(int) [1]	Returns the character at the passed position	Extension	
Enum	startsWith(String,int) [1]	Returns if the string starts with the passed string, from the passed position	Extension	
Enum	length() [1]	Returns the length of the string	Extension	
Enum	equals(String) [1]	Returns if the string is equals to the passed string	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
Enum	trim() [1]	Returns a trimmed version of the string	Extension	

- [1] - This method is only available if java.lang.Enum is stored as String (the Enum name) instead of the numeric value (ordinal value).
- [2] - This method is only available for some databases, such as DB2, Oracle and Postgresql.

Here are a few examples using some of the extensions available for RDBMS. Please refer to the JDOQL Methods Guide for more examples of the JDO standard methods.

Example 1 - Date.getYear() method

Here's an example using the Sale's in a store, where all Sale's are persisted with a date. We want to find all sales in a particular year. Clearly we dont have a field called "year", but we can make use of the DataNucleus extension "Date.getYear()" method

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Sale.class);
query.declareParameters("int theYear");
query.setFilter("date.getYear() == theYear");
List results = (List)query.execute(new Integer(2004));

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Sale " +
                "WHERE date.getYear() == theYear" +
                "PARAMETERS int theYear");
List results = (List)query.execute(new Integer(2004));
```

Example 2 - Using contains with arrays

Here's an example using the Product and checking if one element is part of the Product composition. Note that Production.composition is of type String[].

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setFilter("this.composition.contains(element)");
query.declareParameters("String element");
List results = (List)query.execute("sugar");
```

Example 3 - Defining arrays

Here's an example of array definition using curly braces {} in the query. In this example we check that the Product can be easily manufactured by verifying the number of elements in the composition. Note that Production.composition is of type String[].

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setFilter("{'Candy','Ice cream'}.contains(this.name) &&
this.composition.length < 5");
List results = (List)query.execute();
```

Example 4 - Analysis.rollup() method

Here's an example using the Product in a store and the rollup method. We want to find the max price of a product and all products at given end dates.

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setResult("name, endDate, max(price)");
query.setGrouping("Analysis.rollup({name,endDate})");
List results = (List)query.execute();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7.4 JDOQL2 : Methods

JDOQL2 : Methods

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. In addition, DataNucleus adds on many extensions to the JDO specifications providing all of the commonly required methods. Below is a list of what is required by JDOQL in JDO 1.0 and JDO 2.0, and what DataNucleus RDBMS supports. **This is for the "JDOQL2" implementation for RDBMS**

Java Type	Method	Description	Specification	DataNucleus support
String	startsWith(String)	Returns if the string starts with the passed string	JDO 1.0, JDO 2.0	
String	endsWith(String)	Returns if the string ends with the passed string	JDO 1.0, JDO 2.0	
String	equals(String)	Returns if the strings are equal	Extension	
String	indexOf(String)	Returns the first position of the passed string	JDO 2.0	
String	indexOf(String, int)	Returns the position of the passed string, after the passed position	JDO 2.0	
String	substring(int)	Returns the substring starting from the passed position	JDO 2.0	
String	substring(int,int)	Returns the substring between the passed positions	JDO 2.0	
String	toLowerCase()	Returns the string in lowercase	JDO 2.0	
String	toUpperCase()	Retuns the string in UPPERCASE	JDO 2.0	
String	matches(String pattern)	Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method.	JDO 2.0	
String	charAt(int)	Returns the character at the passed position	Extension	
String	startsWith(String, int)	Returns if the string starts with the passed string, from the passed position	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
String	length()	Returns the length of the string	Extension	
String	equals(String)	Returns if the string is equals to the passed string	Extension	
String	trim()	Returns a trimmed version of the string	Extension	
String	trimLeft()	Returns a trimmed version of the string (trimmed for leading spaces)	Extension	
String	trimRight()	Returns a trimmed version of the string (trimmed for trailing spaces)	Extension	
Collection	isEmpty()	Returns whether the collection is empty	JDO 1.0, JDO 2.0	
Collection	contains(value)	Returns whether the collection contains the passed element	JDO 1.0, JDO 2.0	
Collection	size()	Returns the number of elements in the collection	JDO 2.0	
Map	isEmpty()	Returns whether the map is empty	JDO 1.0, JDO 2.0	
Map	containsKey(key)	Returns whether the map contains the passed key	JDO 2.0	
Map	containsValue(value)	Returns whether the map contains the passed value	JDO 2.0	
Map	containsEntry(key, value)	Returns whether the map contains the passed entry	Extension	
Map	get(key)	Returns the value from the map with the passed key	JDO 2.0	
Map	size()	Returns the number of entries in the map	JDO 2.0	
Math	abs(number)	Returns the absolute value of the passed number	JDO 2.0	
Math	acos(number)	Returns the arc cosine of the passed number	Extension	
Math	asin(number)	Returns the arc sine of the passed number	Extension	
Math	atan(number)	Returns the arc tangent of the passed number	Extension	
Math	ceil(number)	Returns the ceiling of the passed number	Extension	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
Math	cos(number)	Returns the cosine of the passed number	Extension	✓
Math	exp(number)	Returns the exponent of the passed number	Extension	✓
Math	floor(number)	Returns the floor of the passed number	Extension	✓
Math	log(number)	Returns the log(base e) of the passed number	Extension	✓
Math	sin(number)	Returns the absolute value of the passed number	Extension	✓
Math	sqrt(number)	Returns the square root of the passed number	JDO 2.0	✓
Math	tan(number)	Returns the tangent of the passed number	Extension	✓
Date	getDay()	Returns the day (of the month) for the date	Extension	✓
Date	getMonth()	Returns the month for the date	Extension	✓
Date	getYear()	Returns the year for the date	Extension	✓
Time	getHour()	Returns the hour for the time	Extension	✓
Time	getMinute()	Returns the minute for the time	Extension	✓
Time	getSecond()	Returns the second for the time	Extension	✓
JDOHelper	getObjectId(object)	Returns the object identity of the passed persistent object	JDO 2.0	✓
JDOHelper	getVersion(object)	Returns the object version of the passed persistent object	Extension	✓
SQL	rollup({object})	Perform a rollup operation over the results. Only available for some datastores e.g DB2, MSSQL, Oracle	Extension	✓
SQL	cube({object})	Perform a cube operation over the results. Only available for some datastores e.g DB2, MSSQL, Oracle	Extension	✓

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Java Type	Method	Description	Specification	DataNucleus support
{}	length	Returns the length of an array	Extension	
{}	contains(object)	Returns true if the array contains the object	Extension	
Enum	ordinal	Returns the ordinal of the enum (not implemented for enum expression when persisted as a string)	Extension	
Enum	toString()	Returns the string form of the enum (not implemented for enum expression when persisted as a numeric)	Extension	

Here are a few examples using some of the extensions available for RDBMS with "JDOQL2". Please refer to the JDOQL Methods Guide for more examples of the JDO standard methods.

Example 1 - Date.getYear() method

Here's an example using the Sale's in a store, where all Sale's are persisted with a date. We want to find all sales in a particular year. Clearly we dont have a field called "year", but we can make use of the DataNucleus extension "Date.getYear()" method

```
Declarative JDOQL :
Query query = pm.newQuery(org.datanucleus.samples.store.Sale.class);
query.declareParameters("int theYear");
query.setFilter("date.getYear() == theYear");
List results = (List)query.execute(new Integer(2004));

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Sale " +
                "WHERE date.getYear() == theYear" +
                "PARAMETERS int theYear");
List results = (List)query.execute(new Integer(2004));
```

Example 2 - Using contains with arrays

Here's an example using the Product and checking if one element is part of the Product composition. Note that Production.composition is of type String[].

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setFilter("this.composition.contains(element)");
query.declareParameters("String element");
List results = (List)query.execute("sugar");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Example 3 - Defining arrays

Here's an example of array definition using curly braces {} in the query. In this example we check that the Product can be easily manufactured by verifying the number of elements in the composition. Note that Production.composition is of type String[].

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setFilter("{'Candy','Ice cream'}.contains(this.name) &&
this.composition.length < 5");
List results = (List)query.execute();
```

Example 4 - SQL.rollup() method

Here's an example using the Product in a store and the rollup method. We want to find the max price of a product and all products at given end dates.

```
Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);
query.setResult("name, endDate, max(price)");
query.setGrouping("SQL.rollup({name,endDate})");
List results = (List)query.execute();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7.5 JDOQL Spatial Methods

JDOQL : Spatial Methods

When querying spatial data you can make use of a set of spatial methods on the various Java geometry types. The list contains all of the functions detailed in Section 3.2 of the OGC Simple Features specification. Additionally DataNucleus provides some commonly required methods like bounding box test and datastore specific functions. The following tables list all available functions as well as information about which RDBMS implement them. An entry in the "Result" column indicates, whether the funcion may be used in the result part of a JDOQL query.

Functions for Constructing a Geometry Value given its Well-known Text Representation (OGC SF 3.2.6)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.geomFromText(String, Integer)	Construct a Geometry value given its well-known textual representation.	OGC SF	✗	✓	✓	✓
Spatial.pointFromText(String, Integer)	Construct a Point.	OGC SF	✗	✓	✓	✓
Spatial.lineFromText(String, Integer)	Construct a LineString.	OGC SF	✗	✓	✓	✓
Spatial.polyFromText(String, Integer)	Construct a Polygon.	OGC SF	✗	✓	✓	✓
Spatial.mPointFromText(String, Integer)	Construct a MultiPoint.	OGC SF	✗	✓	✓	✓
Spatial.mLineFromText(String, Integer)	Construct a MultiLineString.	OGC SF	✗	✓	✓	✓
Spatial.mPolyFromText(String, Integer)	Construct a MultiPolygon.	OGC SF	✗	✓	✓	✓
Spatial.geomCollFromText(String, Integer)	Construct a GeometryCollection.	OGC SF	✗	✓	✓	✓

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

Functions for Constructing a Geometry Value given its Well-known Binary Representation (OGC SF 3.2.7)

 ALL RIGHTS RESERVED

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.geomFromWKB(String, Integer)	Construct a Geometry value given its well-known binary representation.	OGC SF	✗	✓	✓	✓
Spatial.pointFromWKB(String, Integer)	Construct a Point.	OGC SF	✗	✓	✓	✓
Spatial.lineFromWKB(String, Integer)	Construct a LineString.	OGC SF	✗	✓	✓	✓
Spatial.polyFromWKB(String, Integer)	Construct a Polygon.	OGC SF	✗	✓	✓	✓
Spatial.mPointFromWKB(String, Integer)	Construct a MultiPoint.	OGC SF	✗	✓	✓	✓
Spatial.mLineFromWKB(String, Integer)	Construct a MultiLineString.	OGC SF	✗	✓	✓	✓
Spatial.mPolyFromWKB(String, Integer)	Construct a MultiPolygon.	OGC SF	✗	✓	✓	✓
Spatial.geomCollFromWKB(String, Integer)	Construct a GeometryCollection.	OGC SF	✗	✓	✓	✓

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

Functions on Type Geometry

(OGC SF 3.2.10)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.dimension(Geometry)	Returns the dimension of the Geometry.	OGC SF	✓	✓	✓	✓
Spatial.geometryType(Geometry)	Returns the name of the instantiable subtype of Geometry.	OGC SF	✓	✓	✓	✓
Spatial.asText(Geometry)	Returns the well-known textual representation.	OGC SF	✓	✓	✓	✓
Spatial.asBinary(Geometry)	Returns the well-known binary representation.	OGC SF	✗	✓	✓	✓
Spatial.srid(Geometry)	Returns the Spatial Reference System ID for this Geometry.	OGC SF	✓	✓	✓	✓
Spatial.isEmpty(Geometry)	TRUE if this Geometry corresponds to the empty set.	OGC SF	⚠ [1]	✓	✗ [2]	✓

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.isSimple(Geometry)	TRUE if this Geometry is simple, as defined in the Geometry Model.	OGC SF	[1]		[2]	
Spatial.boundary(Geometry)	Returns a Geometry that is the combinatorial boundary of the Geometry.	OGC SF			[2]	
Spatial.envelope(Geometry)	Returns the rectangle bounding Geometry as a Polygon.	OGC SF				

[1] Oracle does not allow boolean expressions in the SELECT-list.
[2] MySQL does not implement these functions.

Functions on Type Point
(OGC SF 3.2.11)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.x(Point)	Returns the x-coordinate of the Point as a Double.	OGC SF				
Spatial.y(Point)	Returns the y-coordinate of the Point as a Double.	OGC SF				

Functions on Type Curve
(OGC SF 3.2.12)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.startPoint(Curve)	Returns the first point of the Curve.	OGC SF				
Spatial.endPoint(Curve)	Returns the last point of the Curve.	OGC SF				
Spatial.isRing(Curve)	Returns TRUE if Curve is closed and simple. .	OGC SF	[1]		[2]	

[1] Oracle does not allow boolean expressions in the SELECT-list.
[2] MySQL does not implement these functions.

Functions on Type Curve and Type MultiCurve
(OGC SF 3.2.12, 3.2.17)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.isClosed(Curve) Spatial.isClosed(MultiCurve)	Returns TRUE if Curve is closed, i.e., if StartPoint(Curve) = EndPoint(Curve).	OGC SF	⚠ [1]	✔	✔	✔
Spatial.length(Curve) Spatial.length(MultiCurve)	Returns the length of the Curve.	OGC SF	✔	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type LineString
(OGC SF 3.2.13)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.numPoints(LineString)	Returns the number of points in the LineString.	OGC SF	✔	✔	✔	✔
Spatial.pointN(LineString, Integer)	Returns Point n.	OGC SF	✔	✔	✔	✔

Functions on Type Surface and Type MultiSurface
(OGC SF 3.2.14, 3.2.18)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.centroid(Surface) centroid(MultiSurface)	Returns the centroid of Surface, which may lie outside of it.	OGC SF	✔	✔	✘ [1]	✔
Spatial.pointOnSurface(Surface) pointOnSurface(MultiSurface)	Returns a Point guaranteed to lie on the surface.	OGC SF	✔	✔	✘ [1]	✔
Spatial.area(Surface) area(MultiSurface)	Returns the area of Surface.	OGC SF	✔	✔	✔	✔

[1] MySQL does not implement these functions.

Functions on Type Polygon
(OGC SF 3.2.15)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.exteriorRing(Polygon)	Returns the exterior ring of Polygon.	OGC SF	✔	✔	✔	✔

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.numInteriorRing(Polygon)	Returns the number of interior rings.	OGC SF				
Spatial.interiorRingN(Polygon, Integer)	Returns the nth interior ring.	OGC SF				

Functions on Type GeomCollection
(OGC SF 3.2.16)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.numGeometries(GeomCollection)	Returns the number of geometries in the collection.	OGC SF				
Spatial.geometryN(GeomCollection, Integer)	Returns the nth geometry in the collection.	OGC SF				

Functions that test Spatial Relationships
(OGC SF 3.2.19)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.equals(Geometry, Geometry)	TRUE if the two geometries are spatially equal.	OGC SF			[2]	
Spatial.disjoint(Geometry, Geometry)	TRUE if the two geometries are spatially disjoint.	OGC SF			[2]	
Spatial.touches(Geometry, Geometry)	TRUE if the first Geometry spatially touches the other Geometry.	OGC SF			[2]	
Spatial.within(Geometry, Geometry)	TRUE if first Geometry is completely contained in second Geometry.	OGC SF			[2]	
Spatial.overlaps(Geometry, Geometry)	TRUE if first Geometries is spatially overlapping the other Geometry.	OGC SF			[2]	
Spatial.crosses(Geometry, Geometry)	TRUE if first Geometry crosses the other Geometry.	OGC SF			[3]	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.intersects(Geometry, Geometry)	TRUE, if first Geometry spatially intersects the other Geometry.	OGC SF			[2]	
Spatial.contains(Geometry, Geometry)	TRUE, if second Geometry is completely contained in first Geometry.	OGC SF			[2]	
Spatial.relate(Geometry, Geometry, String)	TRUE if the spatial relationship specified by the patternMatrix holds.	OGC SF			[3]	

[1] Oracle does not allow boolean expressions in the SELECT-list.
[2] MySQL does not implement these functions according to the specification.They return the same result as the corresponding MBR-based functions.
[3] MySQL does not implement these functions.

Function on Distance Relationships

(OGC SF 3.2.20)

Method	Description	Specification	Result	PostGIS	MySQL	Oracle Spatial
Spatial.distance(Geometry, Geometry)	Returns the distance between the two geometries.	OGC SF			[1]	

[1] MySQL does not implement this function.

Functions that implement Spatial Operators

(OGC SF 3.2.21)

Method	Description	Specification	Result	PostGIS	MySQL [1]	Oracle Spatial
Spatial.intersection(Geometry, Geometry)	Returns a Geometry that is the set intersection of the two geometries.	OGC SF				
Spatial.difference(Geometry, Geometry)	Returns a Geometry that is the closure of the set difference of the two geometries.	OGC SF				
Spatial,union(Geometry, Geometry)	Returns a Geometry that is the set union of the two geometries.	OGC SF				

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Description	Specification	Result	PostGIS	MySQL [1]	Oracle Spatial
Spatial.symDifference(Geometry, Geometry)	Returns a Geometry that is the closure of the set symmetric difference of the two geometries.	OGC SF				
Spatial.buffer(Geometry, Double)	Returns a Geometry defined by buffering a distance around the Geometry.	OGC SF				
Spatial.convexHull(Geometry)	Returns a Geometry that is the convex hull of the Geometry.	OGC SF				

[1] These functions are currently not implemented in MySQL.They may appear in future releases.

Test whether the bounding box of one geometry intersects the bounding box of another

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
Spatial.bboxTest(Geometry, Geometry)	Returns TRUE if if the bounding box of the first Geometry overlaps second Geometry's bounding box	[1]			

[1] Oracle does not allow boolean expressions in the SELECT-list.

PostGIS Spatial Operators

These functions are only supported on PostGIS.

Method	Description	Result
PostGIS.bboxOverlapsLeft(Geometry, Geometry)	The PostGIS &< operator returns TRUE if the bounding box of the first Geometry overlaps or is to the left of second Geometry's bounding box	
PostGIS.bboxOverlapsRight(Geometry, Geometry)	The PostGIS &> operator returns TRUE if the bounding box of the first Geometry overlaps or is to the right of second Geometry's bounding box	
PostGIS.bboxLeft(Geometry, Geometry)	The PostGIS << operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the left of second Geometry's bounding box	
PostGIS.bboxRight(Geometry, Geometry)	The PostGIS >> operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the right of second Geometry's bounding box	

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Description	Result
PostGIS.bboxOverlapsBelow(Geometry, Geometry)	The PostGIS &<@ operator returns TRUE if the bounding box of the first Geometry overlaps or is below second Geometry's bounding box	
PostGIS.bboxOverlapsAbove(Geometry, Geometry)	The PostGIS |&> operator returns TRUE if the bounding box of the first Geometry overlaps or is above second Geometry's bounding box	
PostGIS.bboxBelow(Geometry, Geometry)	The PostGIS <<| operator returns TRUE if the bounding box of the first Geometry is strictly below second Geometry's bounding box	
PostGIS.bboxAbove(Geometry, Geometry)	The PostGIS |>> operator returns TRUE if the bounding box of the first Geometry is strictly above second Geometry's bounding box	
PostGIS.sameAs(Geometry, Geometry)	The PostGIS ~= operator returns TRUE if the two geometries are vertex-by-vertex equal.	
PostGIS.bboxWithin(Geometry, Geometry)	The PostGIS @ operator returns TRUE if the bounding box of the first Geometry overlaps or is completely contained by second Geometry's bounding box	
PostGIS.bboxContains(Geometry, Geometry)	The PostGIS ~ operator returns TRUE if the bounding box of the first Geometry completely contains second Geometry's bounding box	

MySQL specific Functions for Testing Spatial Relationships between Minimal Bounding Boxes

These functions are only supported on MySQL.

Method	Result
MySQL.mbrEqual(Geometry, Geometry)	
MySQL.mbrDisjoint(Geometry, Geometry)	
MySQL.mbrIntersects(Geometry, Geometry)	
MySQL.mbrTouches(Geometry, Geometry)	
MySQL.mbrWithin(Geometry, Geometry)	
MySQL.mbrContains(Geometry, Geometry)	
MySQL.mbrOverlaps(Geometry, Geometry)	

Oracle specific Functions for Constructing SDO_GEOMETRY types

These functions are only supported on Oracle Spatial.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Method	Desription
Oracle.sdo_geometry(Integer gtype, Integer srid, SDO_POINT point, SDO_ELEM_INFO_ARRAY elem_info, SDO_ORDINATE_ARRAY ordinates)	Creates a SDO_GEOMETRY geometry from the passed geometry type, srid, point, element infos and ordinates.
Oracle.sdo_point_type(Double x, Double y, Double z)	Creates a SDO_POINT geometry from the passed ordinates.
Oracle.sdo_elem_info_array(String numbers)	Creates a SDO_ELEM_INFO_ARRAY from the passed comma-separeted integers.
Oracle.sdo_ordinate_array(String ordinates)	Creates a SDO_ORDINATE_ARRAY from the passed comma-separeted doubles.

Examples

The following sections provide some examples of what can be done using spatial methods in JDOQL queries. In the examples we use a class from the test suite. Here's the source code for reference:

```
package org.datanucleus.samples.pggeometry;

import org.postgis.LineString;

public class SampleLineString {
    private long id;
    private String name;
    private LineString geom;

    public SampleLineString(long id, String name, LineString lineString) {
        this.id = id;
        this.name = name;
        this.geom = lineString;
    }

    public long getId() {
       return id;
    }
    ....
}
```

```
<jdo>
    <package name="org.datanucleus.samples.pggeometry">
        <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/>
                <extension vendor-name="datanucleus" key="spatial-srid"
value="4326"/>

                <class name="SampleLineString" table="samplepglinestring"
detachable="true">
                        <field name="id"/>
                        <field name="name"/>
                        <field name="geom" persistence-modifier="persistent"/>
                </class>
        </package>
</jdo>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Example 1 - Spatial Function in the Filter of a Query

This example shows how to use spatial functions in the filter of a query. The query returns a list of `SampleLineStrings` whose line string has a length less than the given limit.

```
Double limit = new Double(100.0);
Query query = pm.newQuery(SampleLineString.class, "geom != null &amp;&amp;
Spatial.length(geom) < :limit");
List list = (List) query.execute(limit);
```

Example 2 - Spatial Function in the Result Part of a Query

This time we use a spatial function in the result part of a query. The query returns the length of the line string from the selected `SampleLineString`

```
query = pm.newQuery(SampleLineString.class, "id == :id");
query.setResult("Spatial.pointN(geom, 2)");
query.setUnique(true);
Geometry point = (Geometry) query.execute(new Long(1001));
```

Example 3 - Nested Functions

You may want to use nested functions in your query. This example shows how to do that. The query returns a list of `SampleLineStrings`, whose end point spatially equals a given point.

```
Point point = new Point("SRID=4326;POINT(110 45)");
Query query = pm.newQuery(SampleLineString.class, "geom != null &&
Spatial.equals(Spatial.endPoint(geom), :point)");
List list = (List) query.execute(point);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.8 Schema

Datastore Schema

Java persistence tools often add various types of information to the tables for persisted classes, such as special columns, or meta information. DataNucleus is very unobtrusive as far as the datastore schema is concerned. It minimises the addition of any implementation artifacts to the datastore, and adds nothing (other than any datastore identities, and version columns where requested) to any schema tables. The only thing that DataNucleus could add to the schema is a single table that stores information about the classes being persisted (the "auto start mechanism" *SchemaTable* which is optional).

DataNucleus can persist to a datastore that already contains a schema, or alternatively can create the datastore schema for you. It can exist alongside existing tables leaving them untouched. It is not necessary to have a datastore schema specifically for DataNucleus.

When having DataNucleus generate the schema for you, it is advisable to run the DataNucleus SchemaTool before running your application. This can be used to ensure that the correct datastore schema is present (a schema that matches your metadata), and so the persistence layer of your application is ready to use.

Schema Creation

DataNucleus can generate the schema for the developer as it encounters classes in the persistence process. Alternatively you can use the SchemaTool application before starting your application.

If you want to create the schema during the persistence process, the property datanucleus.autoCreateSchema provides a way of telling DataNucleus to do this. Thereafter, during calls to DataNucleus to persist classes or perform queries of persisted data, whenever it encounters a new class to persist that it has no information about, it will use the MetaData to check the datastore for presence of the table, and if it doesn't exist, will create it. In addition it will validate the correctness of the table (compared to the JDO Meta-Data for the class), and any foreign key constraints that it requires (to manage any relationships). If any foreign key constraints are missing it will create them.

If you wanted to only create the tables required, and none of the constraints the property datanucleus.autoCreateTables provides this, simply performing the tables part of the above.

If you want to create any missing columns that are required, the property datanucleus.autoCreateColumns provides this, validating and adding any missing columns.

If you wanted to only create the constraints required, and none of the tables the property datanucleus.autoCreateConstraints provides this, simply performing the constraints part of the above.

Schema Validation

DataNucleus can check any existing schema against what is implied by the JDO Meta-Data.

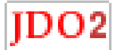
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The property datanucleus.validateTables provides a way of telling DataNucleus to validate any tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the JDO Meta-Data definition) they would set this property to true. This can be useful for example where you are trying to map to an existing schema and want to verify that you've got the correct JDO Meta-Data definition.

The property datanucleus.validateColumns provides a way of telling DataNucleus to validate any columns of the tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the JDO Meta-Data definition) they would set this property to true. This will validate the precise column types and widths etc, including defaultability/nullability settings. **Please be aware that many JDBC drivers contain bugs that return incorrect column detail information and so having this turned off is sometimes the only option (dependent on the JDBC driver quality).**

The property datanucleus.validateConstraints provides a way of telling DataNucleus to validate any constraints (primary keys, foreign keys, indexes) that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their table constraints match what DataNucleus requires (from the JDO Meta-Data definition) they would set this property to true.

Schema Naming Issues



DataNucleus will, by default, use the default database schema for the Connection URL and user supplied. This may cause issues where the user has been set up and in some databases (e.g Oracle) you want to write to a different schema (which that user has access to). To achieve this in DataNucleus you would set the runtime properties

```
javax.jdo.mapping.Catalog={the_catalog_name}
javax.jdo.mapping.Schema={the_schema_name}
```

This will mean that all RDBMS DDL and SQL statements will prefix table names with the necessary catalog and schema names (specify which ones your datastore supports).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.9 SchemaTool

RDBMS : SchemaTool

The meta-data files and annotations define how a class maps across to a persistent store (e.g database). The database schema can be

- Already existing, and so the user maps their classes to their existing tables
- Manually created by the user.
- Automatically created at runtime by use of the datanucleus.autoCreateSchema property.
- Created before running the application, using DataNucleus SchemaTool.

We will describe here the use of DataNucleus SchemaTool. It's included in the DataNucleus RDBMS jar, and is very simple to operate.

DataNucleus SchemaTool has the following modes of operation :

- create - create all database tables required for the classes defined by the input data.
- delete - delete all database tables required for the classes defined by the input data.
- validate - validate all database tables required for the classes defined by the input data.
- dbinfo - provide detailed information about the database, it's limits and datatypes support.
- schemainfo - provide detailed information about the database schema.

In addition, the create mode can be used by adding "-ddlFile {filename}" and this will then not create the schema, but instead output the DDL for the tables into the specified file.

For the **create**, **delete** and **validate** modes DataNucleus SchemaTool accepts either of the following types of input.

- A set of MetaData and class files. The MetaData files define the persistence of the classes they contain. The class files are provided when the classes have annotations.
- The name of a **persistence-unit**. The persistence-unit name defines all classes, metadata files, and jars that make up that unit. Consequently, running DataNucleus SchemaTool with a persistence unit name will create the schema for all classes that are part of that unit.

Here we provide many different ways to invoke DataNucleus SchemaTool

- Invoke it manually from the command line
- Invoke it using Maven1, with the DataNucleus Maven1 plugin
- Invoke it using Maven2, with the DataNucleus Maven2 plugin
- Invoke it using Ant, using the provided DataNucleus SchemaTool ant task
- Invoke it using Jython
- Invoke it using the DataNucleus Eclipse plugin
- Invoke it programmatically from within an application

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Manual Usage

If you wish to call DataNucleus SchemaTool manually, it can be called as follows

```
java [-cp classpath] [system_props] org.datanucleus.store.rdbms.SchemaTool [modes]
[options] [props]
                    [jdo-files] [class-files]
    where system_props (when specified) should include
        -Ddatanucleus.ConnectionDriverName=db_driver_name
        -Ddatanucleus.ConnectionURL=db_url
        -Ddatanucleus.ConnectionUserName=db_username
        -Ddatanucleus.ConnectionPassword=db_password
        -Ddatanucleus.Mapping=orm_mapping_name (optional)
        -Dlog4j.configuration=file:{log4j.properties} (optional)
    where modes can be
        -create : Create the tables specified by the jdo-files/class-files
        -delete : Delete the tables specified by the jdo-files/class-files
        -validate : Validate the tables specified by the jdo-files/class-files
        -dbinfo : Detailed information about the database
        -schemainfo : Detailed information about the database schema
    where options can be
        -ddlFile {filename} : only for use with "create" mode to dump the DDL to the
specified file
        -completeDdl : when using "ddlFile" to get all DDL output and not just
missing tables/constraints
        -api : The API that is being used (default is JDO, but can be set to JPA)
        -persistenceUnit {persistence-unit-name} : Name of the persistence unit to
manage the schema for
        -v : verbose output
    where props can be
        -props {propsfilename} : PMF properties to use in place of the
"system_props"
```

All classes, MetaData files, "persistence.xml" files must be present in the CLASSPATH. In terms of the schema to use, you either specify the "props" file (recommended), or you specify the System properties defining the database connection. You should only specify one of the [modes] above. Let's make a specific example and see the output from SchemaTool. So we have the following files in our application

```
src/java/...   (source files and MetaData files)
target/classes/...    (enhanced classes, and MetaData files)
lib/log4j.jar
lib/datanucleus-core.jar
lib/datanucleus-rdbms.jar
lib/jdo2-api.jar
lib/mysql-connector-java.jar    (JDBC driver for our database)
log4j.properties
```

So we want to create the schema for our persistent classes. So let's invoke DataNucleus SchemaTool to do this, from the top level of our project. In this example we're using Linux (change the CLASSPATH definition to suit for Windows)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
java -cp
target/classes:lib/log4j.jar:lib/jdo2-api.jar:lib/datanucleus-core.jar:lib/datanucleus-rdbms.jar:
                lib/mysql-connector-java.jar
      -Dlog4j.configuration=file:log4j.properties
      org.datanucleus.store.rdbms.SchemaTool -create
      -props datanucleus.properties
      target/classes/org/datanucleus/examples/normal/package.jdo
      target/classes/org/datanucleus/examples/inverse/package.jdo


DataNucleus SchemaTool (version 1.0.0) : Creation of the schema

DataNucleus SchemaTool : Classpath
>>  /home/andy/work/DataNucleus/samples/packofcards/target/classes
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/log4j.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-core.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-rdbms.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/jdo2-api.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/mysql-connector-java.jar

DataNucleus SchemaTool : Input Files
>>
/home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/examples/inverse/package.j
>>
/home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/examples/normal/package.jd

DataNucleus SchemaTool : Taking JDO properties from file "datanucleus.properties"

SchemaTool completed successfully
```

So as you see, DataNucleus SchemaTool prints out our input, the properties used, and finally a success message. If an error occurs, then something will be printed to the screen, and more information will be written to the log.

Maven1

If you are using Maven1 to build your system, you will need the DataNucleus Maven plugin. This provides 3 goals representing the different modes of DataNucleus SchemaTool. You can use the goals datanucleus:schema-create, datanucleus:schema-delete, datanucleus:schema-validate depending on whether you want to create, delete or validate the database tables. To use the DataNucleus Maven1 plugin you will need to set properties for the plugin (in your project.properties). For example

```
maven.datanucleus.verbose=true
maven.datanucleus.schematool.ddlfile=
maven.datanucleus.orm.mapping=mysql
maven.datanucleus.log4j.configuration=file:log4j.properties
maven.datanucleus.properties=${basedir}/datanucleus.properties
```

So with these example properties I am setting the Maven plugin to use PMF properties from the file "datanucleus.properties" at the root of the Maven project. I am also specifying a log4j configuration file

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

defining the logging for the SchemaTool process. In addition, I am specifying my object-relational mapping (O/R mapping) in files with names like package-mysql.orm. This last part is optional, and you can specify the mapping information in the JDO MetaData files. The other property we have defined is to receive verbose output from SchemaTool. The output obtained from Maven like this is identical to that obtained when invoking the SchemaTool manually.

Maven2

If you are using Maven2 to build your system, you will need the DataNucleus Maven2 plugin. This provides 5 goals representing the different modes of DataNucleus SchemaTool. You can use the goals datanucleus:schema-create, datanucleus:schema-delete, datanucleus:schema-validate depending on whether you want to create, delete or validate the database tables. To use the DataNucleus Maven2 plugin you will may need to set properties for the plugin (in your pom.xml). For example

Property	Default	Description
mappingIncludes	**/*.jdo	Fileset to include for schema generation
mappingExcludes		Fileset to exclude for schema generation
persistenceUnitName		Name of the persistence-unit to generate the schema for
props		Name of a properties file for the datastore (PMF)
log4jConfiguration		Config file location for Log4J (if using it)
jdkLogConfiguration		Config file location for JDK1.4 logging (if using it)
api	JDO	API to enhance to (JDO, JPA)
verbose	false	Verbose output?
fork	true	Whether to fork the enhancer process
outputFile		Name of an output file to dump any DDL to
completeDdl	false	Whether to generate DDL including things that already exist?

So to give an example, I add the following to my *pom.xml*

```
    <build>
        ...
        <plugins>
            <plugin>
                <groupId>org.datanucleus</groupId>
                <artifactId>datanucleus-maven-plugin</artifactId>
                <version>1.1.0</version>
                <configuration>
                    <props>${basedir}/datanucleus.properties</props>
<log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
                    <verbose>true</verbose>
                </configuration>
            </plugin>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        </plugins>
        ...
    </build>
```

So with these properties when I run SchemaTool it uses properties from the file *datanucleus.properties* at the root of the Maven project. I am also specifying a log4j configuration file defining the logging for the SchemaTool process. I then can invoke any of the Maven2 goals

```
mvn datanucleus:schema-create             Create the Schema
mvn datanucleus:schema-delete             Delete the schema
mvn datanucleus:schema-validate           Validate the Schema
mvn datanucleus:schema-info               Output info for the Schema
mvn datanucleus:schema-dbinfo             Output info for the datastore
```

Ant

An Ant task is provided for using DataNucleus SchemaTool. It has classname org.datanucleus.store.rdbms.SchemaToolTask, and accepts the following parameters

Parameter	Description	values
mode	Mode of operation.	create, delete, validate, dbinfo, schemainfo
verbose	Whether to give verbose output.	true, **false**
props	The filename to use for PMF properties	
ddlFile	The filename where SchemaTool should output the DDL.	
completeDdl	Whether to output complete DDL (instead of just missing tables). Only used with ddlFile	true, **false**
api	API that we are using in our use of DataNucleus	**JDO** | JPA
persistenceUnit	Name of the persistence-unit that we should manage the schema for	

The SchemaTool task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the SchemaTool task.

In addition to the parameters that the Ant task accepts, you will need to set up your CLASSPATH to include the classes and JDO MetaData files, and to define the following system properties via the sysproperty parameter (not required when specifying the PMF props via the properties file)

Parameter	Description	Optional
datanucleus.ConnectionDriverName	Name of JDBC driver class	Mandatory

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Parameter	Description	Optional
datanucleus.ConnectionURL	URL for the database	Mandatory
datanucleus.ConnectionUserName	User name for the database	Mandatory
datanucleus.ConnectionPassword	Password for the database	Mandatory
datanucleus.Mapping	ORM Mapping name	Optional
log4j.configuration	Log4J configuration file, for SchemaTool's Log	Optional

So you could define something *like* the following, setting up the parameters **schematool.classpath**, **datanucleus.ConnectionDriverName**, **datanucleus.ConnectionURL**, **datanucleus.ConnectionUserName**, and **datanucleus.ConnectionPassword** to suit your situation.

You define the jdo files to create the tables using **fileset**.

```
<taskdef name="schematool" classname="org.datanucleus.store.rdbms.SchemaToolTask" />

<schematool failonerror="true" verbose="true" mode="create">
    <classpath>
        <path refid="schematool.classpath"/>
    </classpath>
    <fileset dir="${classes.dir}">
        <include name="**/*.jdo"/>
    </fileset>
    <sysproperty key="datanucleus.ConnectionDriverName"
                 value="${datanucleus.ConnectionDriverName}"/>
    <sysproperty key="datanucleus.ConnectionURL"
                 value="${datanucleus.ConnectionURL}"/>
    <sysproperty key="datanucleus.ConnectionUserName"
                 value="${datanucleus.ConnectionUserName}"/>
    <sysproperty key="datanucleus.ConnectionPassword"
                 value="${datanucleus.ConnectionPassword}"/>
    <sysproperty key="datanucleus.Mapping"
                 value="${datanucleus.Mapping}"/>
</schematool>
```

Jython

Jython is a powerful scripting language that allows you to automate tasks and easy the development. If you are using Jython, and wants to use DataNucleus tools, all you need is to the place the DataNucleus jars, the persistent classes, metadata files, jdbc driver jars and dependencies into the classpath.

The Jython script may be written in several forms but achieving the same goals.

Here we have a template for invoking the main method of the Schema Tool. The main method acts like the command line, by parsing arguments and invoking the appropriate methods.

```
from org.datanucleus.store.rdbms import SchemaTool
tool = SchemaTool()
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
tool.main(<<<[options] [jdo-files]>>>)
```

The below is a concrete example.

```
from org.datanucleus.store.rdbms import SchemaTool
tool = SchemaTool()
tool.main(["-create",
           "-props=/home/JDOproperties.properties",
           "target/classes/org/datanucleus/examples/normal/package.jdo",
           "target/classes/org/datanucleus/examples/inverse/package.jdo"])
```

For other operations of the SchemaTool consult the DataNucleus javadocs. For questions about Jython, please refer to Jython WebSite.

SchemaTool API

DataNucleus SchemaTool can also be called programmatically from an application. The API is shown below.

```
package org.datanucleus.store.rdbms;

public class SchemaTool
{
     public int createSchema(PersistenceManagerFactory pmf, List classNames)

     public int deleteSchema(PersistenceManagerFactory pmf, List classNames)

     public int validateSchema(PersistenceManagerFactory pmf, List classNames)

}
```

So for example to create the schema for classes *mydomain.A* and *mydomain.B* you would do

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
...
List classNames = new ArrayList();
classNames.add("mydomain.A");
classNames.add("mydomain.B");
try
{
    org.datanucleus.store.rdbms.SchemaTool schematool = new
org.datanucleus.store.rdbms.SchemaTool();
    schemaTool.createSchema(pmf, classNames);
}
catch(Exception e)
{
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

14.2.10 Statement Batching

RDBMS Statement Batching

When changes are required to be made to an underlying RDBMS datastore, statements are sent via JDBC. A statement is, in general, a single SQL command, and is then executed. In some circumstances the statements due to be sent to the datastore are the same JDBC statement several times. In this case the statement can be *batched*. This means that a statement is created for the SQL, and it is passed to the datastore with multiple sets of values before being executed. When it is executed the SQL is executed for each of the sets of values. DataNucleus allows statement batching under certain circumstances.

The maximum number of statements that can be included in a *batch* can be set via a persistence property **datanucleus.rdbms.statementBatchLimit**. This defaults to 50. If you set it to -1 then there is no maximum limit imposed. Setting it to 0 means that batching is turned off.

It should be noted that while batching sounds essential, it is only of any possible use when the exact same SQL is required to be executed more than 1 times in a row. If a different SQL needs executing between 2 such statements then no batching is possible anyway. . Let's take an example

```
INSERT INTO MYTABLE VALUES(?,?,?,?)
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
```

In this example the first two statements can be batched together since they are identical and nothing else separates them. All subsequent statements cannot be batched since no two identical statements follow each other.

The statements that DataNucleus currently allows for batching are

- Delete of objects
- Insert of container elements/keys/values
- Delete of container elements/keys/values

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.11 Views

RDBMS Views

DataNucleus supports persisting objects to RDBMS datastores, persisting to Tables. The majority of RDBMS also provide support for Views, providing the equivalent of a read-only SELECT across various tables. DataNucleus also provides support for querying such Views. This provides more flexibility to the user where they have data and need to display it in their application. Support for Views is described below.

When you want to access data according to a View, you are required to provide a class that will accept the values from the View when queried, and Meta-Data for the class that defines the View and how it maps onto the provided class. Let's take an example. We have a View SALEABLE_PRODUCT in our database as follows, defined based on data in a PRODUCT table.

```
CREATE VIEW SALEABLE_PRODUCT (ID, NAME, PRICE, CURRENCY) AS
    SELECT ID, NAME, CURRENT_PRICE AS PRICE, CURRENCY FROM PRODUCT
    WHERE PRODUCT.STATUS_ID = 1
```

So we define a class to receive the values from this View.

```
package org.datanucleus.samples.views;
public class SaleableProduct
{
    String id;
    String name;
    double price;
    String currency;

    public String getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public double getPrice()
    {
        return price;
    }

    public String getCurrency()
    {
        return currency;
    }
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

and then we define how this class is mapped to the View

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="org.datanucleus.samples.views">
        <class name="SaleableProduct" identity-type="nondurable"
table="SALEABLE_PRODUCT">
            <field name="id"/>
            <field name="name"/>
            <field name="price"/>
            <field name="currency"/>

            <!-- This is the "generic" SQL92 version of the view. -->
            <extension vendor-name="datanucleus" key="view-definition" value="
CREATE VIEW SALEABLE_PRODUCT
(
    {this.id},
    {this.name},
    {this.price},
    {this.currency}
) AS
SELECT ID, NAME, CURRENT_PRICE AS PRICE, CURRENCY FROM PRODUCT
WHERE PRODUCT.STATUS_ID = 1"/>
        </class>
    </package>
</jdo>
```

Please note the following

- We've defined our class as using "nondurable" identity. This is an important step since rows of the View typically don't operate in the same way as rows of a Table, not mapping onto a persisted updateable object as such
- We've specified the "table", which in this case is the view name - otherwise DataNucleus would create a name for the view based on the class name.
- We've defined a DataNucleus extension view-definition that defines the view for this class. If the view doesn't already exist it doesn't matter since DataNucleus (when used with autoCreateSchema) will execute this construction definition.
- The view-definition can contain macros utilising the names of the fields in the class, and hence borrowing their column names (if we had defined column names for the fields of the class).
- You can also utilise other classes in the macros, and include them via a DataNucleus MetaData extension view-imports (not shown here)
- If your View already exists you are still required to provide a view-definition even though DataNucleus will not be utilising it, since it also uses this attribute as the flag for whether it is a View or a Table - just make sure that you specify the "table" also in the MetaData.

We can now utilise this class within normal DataNucleus querying operation.

```
Extent e = pm.getExtent(SaleableProduct.class);
Iterator iter = e.iterator();
while (iter.hasNext())
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    SaleableProduct product = (SaleableProduct)iter.next();
}
```

Hopefully that has given enough detail on how to create and access views from with a DataNucleus-enabled application.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.12 Datastore API

Datastore Schema API

JDO/JPA are APIs for persisting and retrieving objects to/from datastores. They don't provide a way of accessing the schema of the datastore itself (if it has one). In the case of RDBMS it is useful to be able to find out what columns there are in a table, or what data types are supported for example. DataNucleus Access Platform provides an API for this.

The first thing to do is get your hands on the DataNucleus *StoreManager* and from that the *StoreSchemaHandler*. You do this as follows

```
import org.datanucleus.jdo.JDOPersistenceManagerFactory;
import org.datanucleus.store.StoreManager;
import org.datanucleus.store.schema.StoreSchemaHandler;

[assumed to have "pmf"]
...

StoreManager storeMgr = ((JDOPersistenceManagerFactory)pmf).getStoreManager();
StoreSchemaHandler schemaHandler = storeMgr.getSchemaHandler();
```

So now we have the *StoreSchemaHandler* what can we do with it? Well start with the javadoc for the implementation that is used for RDBMS

Datastore Types Information

So we now want to find out what JDBC/SQL types are supported for our RDBMS. This is simple.

```
import org.datanucleus.store.rdbms.schema.RDBMSTypesInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTypesInfo typesInfo = schemaHandler.getSchemaData(conn, "types");
```

As you can see from the javadocs for *RDBMSTypesInfo* we can access the JDBC types information via the "children". They are keyed by the JDBC type number of the JDBC type (see java.sql.Types). So we can just iterate it

```
Iterator jdbcTypesIter = typesInfo.getChildren().values().iterator();
while (jdbcTypesIter.hasNext())
{
    JDBCTypeInfo jdbcType = (JDBCTypeInfo)jdbcTypesIter.next();

    // Each JDBCTypeInfo contains SQLTypeInfo as its children, keyed by SQL name
    Iterator sqlTypesIter = jdbcType.getChildren().values().iterator();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    while (sqlTypesIter.hasNext())
    {
        SQLTypeInfo sqlType = (SQLTypeInfo)sqlTypesIter.next();
        ... inspect the SQL type info
    }
}
```

Column information for a table

Here we have a table in the datastore and want to find the columns present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableInfo tableInfo = schemaHandler.getSchemaData(conn, "columns",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableInfo* we can access the columns information via the "children".

```
Iterator columnsIter = tableInfo.getChildren().iterator();
while (columnsIter.hasNext())
{
    RDBMSColumnInfo colInfo = (RDBMSColumnInfo)columnsIter.next();

    ...
}
```

Index information for a table

Here we have a table in the datastore and want to find the indices present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableIndexInfo tableInfo = schemaHandler.getSchemaData(conn, "indices",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableIndexInfo* we can access the index information via the "children".

```
Iterator indexIter = tableInfo.getChildren().iterator();
while (indexIter.hasNext())
{
    IndexInfo idxInfo = (IndexInfo)indexIter.next();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    ...
}
```

ForeignKey information for a table

Here we have a table in the datastore and want to find the FKs present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableFKInfo tableInfo = schemaHandler.getSchemaData(conn, "foreign-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableFKInfo* we can access the foreign-key information via the "children".

```
Iterator fkIter = tableInfo.getChildren().iterator();
while (fkIter.hasNext())
{
    ForeignKeyInfo fkInfo = (ForeignKeyInfo)fkIter.next();

    ...
}
```

PrimaryKey information for a table

Here we have a table in the datastore and want to find the PK present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTablePKInfo tableInfo = schemaHandler.getSchemaData(conn, "primary-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTablePKInfo* we can access the foreign-key information via the "children".

```
Iterator pkIter = tableInfo.getChildren().iterator();
while (pkIter.hasNext())
{
    PrimaryKeyInfo pkInfo = (PrimaryKeyInfo)pkIter.next();

    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

© 2 0 0 8 - 2 0 0 9 , D a t a N u c l e u s • A L L R I G H T S R E S E R V E D

14.3 DB4O

DB4O Datastores

DataNucleus supports persisting objects to DB4O datastores (using the datanucleus-db4o plugin). Support for DB4O is maturing and will be enhanced further in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

DataNucleus supports 3 modes of operation of *db4o* - file-based, embedded-server-based and client-server based. In order to do so and to fit in with the JDO/JPA APIs we have defined the following means of connection.

The following persistence properties will connect to a local **file-based** DB4O running on your local machine

```
datanucleus.ConnectionURL=db4o:file:{my_db4o_file}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

The filename {my_db4o_file} can be absolute OR relative (and not include the curly brackets!).

The following persistence properties will connect to **embedded-server-based** DB4O running on with a local file

```
datanucleus.ConnectionURL=db4o:server:{my_db4o_file}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

The filename {my_db4o_file} can be absolute OR relative.

The following persistence properties will connect as a client to a **TCP/IP DB4O Server**

```
datanucleus.ConnectionURL=db4o:{db4o_host}:{db4o_port}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DB4O doesn't itself use such URLs so it was necessary to define this DataNucleus-specific way of addressing DB4O.

Known Limitations

The following are known limitations of the current implementation

- DB4O only allows versions to be of type *long*. This means that you cannot use the JDO version strategy of "DATE_TIME".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.3.1 Persistence Properties

DB4O Persistence Properties

DataNucleus accepts a large number of persistence properties for use when defining either a PersistenceManagerFactory or an *EntityManagerFactory*. For DB4O datastores there are several additional properties. These are listed here. Bear in mind that these only work with DataNucleus, and not with any other JDO/JPA implementation.

datanucleus.db4o.outputFile

Description	Name of a file where DB4O will output its log information
Range of Values	

datanucleus.db4o.flushFileBuffers

Description	Whether DB4O should flush its file buffers on commit of the transaction.
Range of Values	**true** | false

datanucleus.db4o.generateUUIDs

Description	Whether DB4O should generate UUIDs for objects of all classes
Range of Values	true | **false**

datanucleus.db4o.lockDatabaseFile

Description	Whether DB4O should lock the database file
Range of Values	**true** | false

datanucleus.db4o.automaticShutdown

Description	Whether DB4O should provide automatic shutdown
Range of Values	**true** | false

datanucleus.db4o.internStrings

Description	Whether DB4O should intern any strings
Range of Values	true | **false**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

datanucleus.db4o.exceptionsOnNotStorable

Description	Whether DB4O should throw an exception on finding a non-storable object
Range of Values	true | **false**

datanucleus.db4o.optimizeNativeQueries

Description	Whether DB4O should optimize native queries
Range of Values	**true** | false

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.3.2 Queries

DB4O Queries

Using a DB4O datastore DataNucleus allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax
- Native - DB4Os own type-safe query language
- SQL - SQL queries of DB4O

It is hoped to provide more options in the future.

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.3.3 Native Queries

DB4O Native Queries

DB4O provides its own "native" query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```
// Find all employees older than 31
Query q = pm.newQuery("Native", new Predicate()
    {
        public boolean match(Person p)
        {
            return p.getAge() >= 32;
        }
    });

List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the DB4O Predicate.

Using Comparators

In DB4O's API you can also specify "comparators" to control the ordering of the returned objects. You can make use of these also using the JDO API. Like this

```
// Find all employees older than 31
Query q = pm.newQuery("Native", new Predicate()
    {
        public boolean match(Person p)
        {
            return p.getAge() >= 32;
        }
    });
q.addExtension("db4o.native.comparator",
    new QueryComparator()
    {
        public int compare(Object o1, Object o2)
        {
            Person p1 = (Person)o1;
            Person p2 = (Person)o2;
            return (p1.getAge() - p2.getAge());
        }
    });

List results = (List)q.execute();
```

So we make use of the query extension **db4o.native.comparator** and pass the comparator in.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.3.4 sql4o

sql4o

DataNucleus provides a useful extension to db4o, known as **sql4o**. This is a project written by Travis Reeder. DataNucleus Access Platform provides an updated version licensed under Apache 2, under agreement with the original author, and utilised by the db4o plugin for SQL queries. The documentation below is taken from the original, but with some enhancements to better clarify some features, and add on any DataNucleus extensions.

Supported SQL Syntax

Clearly to support the full range of ANSI SQL would be a significant task. This plugin supports a reasonable subset of the most useful parts of SQL.

```
SELECT select_expr, ...
FROM object_type
[WHERE where_condition]
[GROUP BY {group_field_name }]
[ORDER BY {order_field_name } [ASC | DESC], ...]
[LIMIT {[offset,] row_count }]
```

where

- **select_expr** reference fields in an object that you would like returned. If used, it will return an array of objects. If SELECT is given and is anything other than *, then the results will be returned as an array of values, NOT the object.
- **object_type** must consist of the fully qualified class name of a single object. (Joins are not yet supported). An alias can be used for example: *FROM ObjectName ob*
- **where_condition** Just as in normal SQL.
- **group_field_name** Just as in normal SQL.
- **order_field_name** Just as in normal SQL.
- **offset, rowcount** To limit the number of records returned

In addition you could select "db4o_id" which is a special keyword supported, to get hold of the internal DB4O object id.

Results

The return values depend on the "select_expr" specified. If select_expr is not present or select_expr equals '*', then all fields of the candidate object will be accessible. If select_expr is present and is not '*', only the fields specified will be accessible.

Queries return a List of *org.datanucleus.sql4o.Result* objects (List<Result>). This List is an instance of *org.datanucleus.sql4o.ObjectSetWrapper* which allows access to some information about number of rows, number of columns in each row and the field names for each column. The Result object allows you to

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

access the return values by using one of the getObject() methods

- Result.getObject(int fieldIndex)
- Result.getObject(String fieldName)

For simple queries where selecting all fields of the candidate class, Result.getBaseObject() will return the underlying object that the field values come from. This will return null for aggregate queries.

Direct SQL Query Execution

The primary way of executing an SQL query is where you have a db4o ObjectContainer. Here we provide the class *Sql4o* with an execute method, allowing direct execution of SQL. Obviously all SQL syntax is not supported, but the principal constraints are. Here's an example

```
ObjectContainer cont = {obtain object container}
...
List results = Sql4o.execute(cont, "SELECT * FROM Contact");
Iterator iter = results.iterator();
while (iter.hasNext())
{
    Object obj = iter.next();
    ...
}
```

SQL Query Execution via JDBC

An alternative to direct execution is where you want to use the familiar JDBC interface. Here is a code sample, obtaining a Connection and execuying a query. The JDBC URL string follows the following styles

- **Server-based** *jdbc:db4o://{HOSTNAME}:{PORT}*
- **File-based** *jdbc:db4o:file:{FILENAME}*

```
// Load Driver
Class.forName("org.datanucleus.sql4o.jdbc.Db4oDriver");

// Get java.sql.Connection
Connection conn = DriverManager.getConnection("jdbc:db4o://localhost:" + port,
                     username, password);

// Create java.sql.Statement from Connection
Statement statement = conn.createStatement();

// Execute query
ResultSet rs = stmt.executeQuery("select * from Contact");

// Iterate over results:
while(rs.next())
{
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println("Got contact: " + name + " age: " + age);
}
```

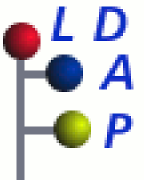
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
// Clean up
rs.close();
statement.close();
conn.close(); // only conn.close is required
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4 LDAP

LDAP Datastores



DataNucleus supports persisting objects to LDAP datastores (using the datanucleus-ldap plugin). Support for LDAP is in its early stages and will be enhanced in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

The following persistence properties will connect to an LDAP running on your local machine

```
datanucleus.ConnectionDriverName=com.sun.jndi.ldap.LdapCtxFactory
datanucleus.ConnectionURL=ldap://localhost:10389
datanucleus.ConnectionUserName=uid=admin,ou=system
datanucleus.ConnectionPassword=secret
```

Connection Pooling

The following persistence properties will enable connection pooling

```
datanucleus.connectionPoolingType=JNDI
```

Once you have turned connection pooling on if you want more control over the pooling you can also set the following persistence properties

- **datanucleus.connectionPool.maxPoolSize** : max size of pool
- **datanucleus.connectionPool.initialPoolSize** : initial size of pool

Known Limitations

The following are known limitations of the current implementation

- Datastore Identity is not currently supported
- Optimistic checking of versions is not supported
- Identity generators that operate using the datastore are not supported
- Cannot map inherited classes to the same LDAP type

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.1 Mapping

LDAP Datastore Mapping

When persisting a Java object to an LDAP datastore clearly the user would like some control over where and how in the LDAP DIT (directory information tree) we are persisting the object. In general Java objects are mapped to LDAP entries and fields of the Java objects are mapped to attributes of the LDAP entries.

Java Types

The following Java types are supported and stored as single-valued attribute to the LDAP entry:

- String, primitives (like int and double), wrappers of primitives (like java.util.Long), java.util.BigDecimal, java.util.BigInteger, java.util.UUID
- boolean and java.lang.Boolean are converted to RFC 4517 "boolean" syntax (TRUE or FALSE)
- java.util.Date and java.util.Calendar are converted to RFC 4517 "generalized time" syntax

Arrays, Collections, Sets and Lists of these data types are stored as multi-valued attributes. Please note that when using Arrays and Lists no order could be guaranteed and no duplicate values are allowed!

Relationships

By default PersistenceCapable objects are stored as separate LDAP entries. There are some options how to persist relationship references between PersistenceCapable objects:

- DN matching
- Attribute matching
- LDAP hierarchies

It is also possible to store PersistenceCapable objects embedded.

Examples

Here's an example using JDO XML MetaData:

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Group" table="ou=Groups,dc=example,dc=com"
schema="top,groupOfNames" detachable="true">
            <field name="name" column="cn" primary-key="true" />
            <field name="users" column="member" />
        </class>

        <class name="Person" table="ou=Users,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson" detachable="true">
            <field name="personNum" column="cn" primary-key="true" />
            <field name="firstName" column="givenMame" />
            <field name="lastName" column="sn" />
        </class>
    </package>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
</jdo>
```

For the class as a whole we use the **table** attribute to set the *distinguished name* of the container under which to store objects of a type. So, for example, we are mapping all objects of class Group as subordinates to "ou=Groups,dc=example,dc=com". You can also use the extension "dn" to specify the same thing.

For the class as a whole we use the **schema** attribute to define the object classes of the LDAP entry. So, for example, all objects of type Person are mapped to the common "top,person,organizationalPerson,inetOrgPerson" object classes in LDAP. You can also use the extension "objectClass" to specify the same thing.

For each field we use the **column** attribute to define the *LDAP attribute* that we are mapping this field to. So, for example, we map the Group "name" to "cn" in our LDAP. You can also use the extension "attribute" to specify the same thing.

Some resulting LDAP entries would look like this:

```
dn: cn=Sales,ou=Groups,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
cn: Sales
member: cn=1,ou=Users,dc=example,dc=com

dn: cn=1,ou=Users,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: 1
givenName: Bugs
sn: Bunny
```

Here's the same example using JDO Annotations:

```
@PersistenceCapable(table = "ou=Groups,dc=example,dc=com", schema =
"top,groupOfNames")
public class Group
{
    @PrimaryKey
    @Column(name = "cn")
    String name;

    @Column(name = "member")
    protected Set<Person> users = new HashSet<Person>();
}

@PersistenceCapable(table = "ou=Users,dc=example,dc=com", schema =
"top,person,organizationalPerson,inetOrgPerson")
public class Person
{
    @PrimaryKey
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    @Column(name = "cn")
    private long personNum;

    @Column(name = "givenName")
    private String firstName;

    @Column(name = "sn")
    private String lastName;
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.2 Relations by DN

Relationship Mapping by DN

A common way to model relationships between LDAP entries is to put the LDAP distinguished name of the referenced LDAP entry to an attribute of the referencing LDAP entry. For example entries with object class groupOfNames use the attribute *member* which contains distinguished names of the group members.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store the relationships.

- Unidirectional
- Bidirectional

1-N Unidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=Departments
|    |-- cn=Sales
|    |-- cn=Engineering
|    |-- ...
|
|-- ou=Employees
|    |-- cn=Bugs Bunny
|    |-- cn=Daffy Duck
|    |-- cn=Speedy Gonzales
|    |-- ...
```

```
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
}
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. The **Employee** knows nothing about the **Department** it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the DN reference could be stored at the one or at the other LDAP entry.

Owner Object Side

The obious way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:

```
dn: cn=Sales,ou=Departments,dc=example,dc=com
objectClass: top
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
objectClass: groupOfNames
cn: Sales
member: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
member: cn=Daffy Duck,ou=Employees,dc=example,dc=com
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees" column="member">
                <extension vendor-name="datanucleus" key="empty-value"
value="uid=admin,ou=system"/>
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>
    </package>
</jdo>
```

So we define that the attribute *member* should be used to persist the relationship of field *employees*.

Note: We use the extension *empty-value* here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.

Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```
dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: cn=Sales,ou=Departments,dc=example,dc=com
```

Our JDO metadata looks like this:

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees">
                <element column="departmentNumber" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>
    </package>
</jdo>
```

We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to. With the *<element>* tag we specify that the relationship should be persisted at the other side, the *column* attribute defines the LDAP attribute to use. In this case the relationship is persisted in the *departmentNumber* attribute at the employee entry.

1-N Bidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=Departments
|   |-- cn=Sales
|   |-- cn=Engineering
|   |-- ...
|
|-- ou=Employees
|   |-- cn=Bugs Bunny
|   |-- cn=Daffy Duck
|   |-- cn=Speedy Gonzales
|   |-- ...
```

```
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Department department;
}
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. Now each **Employee** has a reference to its **Department**.

It is possible to persist this relationship on both sides.

```
dn: cn=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
cn: Sales
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
member: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
member: cn=Daffy Duck,ou=Employees,dc=example,dc=com
```

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees" column="member">
                <extension vendor-name="datanucleus" key="empty-value"
value="uid=admin,ou=system"/>
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department" mapped-by="employees" />
        </class>
    </package>
</jdo>
```

In this case we store the relation at the department entry side in a multi-valued attribute *member*. Now the employee metadata contains a department field that is *mapped-by* the employees field of department.

Note: We use the extension *empty-value* here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.3 Relations by Attribute

Relationship Mapping by Attribute

Another way to model relationships between LDAP entries is to use attribute matching. This means two entries have the same attribute values. An example of this type of relationship is used by posixGroup and posixAccount object classes were posixGroup.memberUid points to posicAccount.uid.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store the relationships.

- Unidirectional
- Bidirectional

1-N Unidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=Departments
|    |-- ou=Sales
|    |-- ou=Engineering
|    |-- ...
|
|-- ou=Employees
|    |-- uid=bbunny
|    |-- uid=dduck
|    |-- uid=sgonzales
|    |-- ...
```

```
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    String uid;
}
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. The **Employee** knows nothing about the **Department** it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the reference could be stored at the one or at the other LDAP entry.

Owner Object Side

One way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:

```
dn: ou=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: organizationalUnit
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
objectClass: extensibleObject
ou: Sales
memberUid: bbunny
memberUid: dduck
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit,extensibleObject">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" column="memberUid">
                <join column="uid" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">

            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
        </class>
    </package>
</jdo>
```

So we define that the attribute *memberUid* at the department entry should be used to persist the relationship of field *employees*

The important thing here is the *<join>* tag and its *column*. Firstly it signals DataNucleus to use attribute mapping. Secondly it specifies the attribute at the other side that should be used for relationship mapping. In our case, when we establish a relationship between a **Department** and an **Employee**, the *uid* value of the employee entry is stored in the *memberUid* attribute of the department entry.

Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```
dn: uid=bbunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: bbunny
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: Sales
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees">
                <element column="departmentNumber" />
                <join column="ou" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
        </class>
    </package>
</jdo>
```

We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to.

With the *<element>* tag we specify that the relationship should be persisted at the other side and the *column* attribute defines the LDAP attribute to use. In this case the relationship is persisted in the *departmentNumber* attribute at the employee entry.

The important thing here is the *<join>* tag and its *column*. As before it signals DataNucleus to use attribute mapping. Now, as the relation is persisted at the other side, it specifies the attribute at this side that should be used for relationship mapping. In our case, when we establish a relationship between a **Department** and an **Employee**, the *ou* value of the department entry is stored in the *departmentNumber* attribute of the employee entry.

1-N Bidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                        public class Department {
|                                                            String name;
|-- ou=Departments                                           Set<Employee> employees;
|   |-- ou=Sales                                         }
|   |-- ou=Engineering
|   |-- ...                                              public class Employee {
|                                                            String firstName;
|-- ou=Employees                                             String lastName;
|   |-- uid=bbunny                                           String fullName;
|   |-- uid=dduck                                            String uid;
|   |-- uid=sgonzales                                        Department department;
|   |-- ...                                              }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. Now each **Employee** has a reference to its **Department**.

It is possible to persist this relationship on both sides.

```
dn: uid=bbunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: bbunny
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: Sales
```

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" mapped-by="department" />
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
            <field name="department" column="departmentNumber">
                <join column="ou" />
            </field>
        </class>
    </package>
</jdo>
```

In this case we store the relation at the employee entry side in a single-valued attribute *departmentNumber*. With the *<join>* tag and its *column* we specify that the *ou* value of the department entry should be used as join value. Also note that *employee* field of **Department** is *mapped-by* the *department* field of the **Employee**.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.4 Relations by Hierarchy

Relationship Mapping by Hierarchy

As LDAP is a hierarchical data store it is possible to model relationships between LDAP entries using hierarchies. For example organisational structures like departments and their employees are often modeled hierarchical in LDAP. It is possible to map 1-1 and N-1/1-N relationships using LDAP hierarchies.

The main challenge with hierarchical mapping is that the distinguished name (DN) of children depends on the DN of their parent. Therefore each child class needs a reference to the parent class. The parent class metadata defines a (fixed) LDAP DN that is used as container for all objects of the parent type. The child class metadata contains a dynamic part in its DN definition. This dynamic part contains the name of the field holding the reference to the parent object, the name is surrounded by curly braces. This dynamic DN is the indicator for DataNucleus to use hierarchical mapping. The reference field itself won't be persisted as attribute because it is used as dynamic parameter. If you query for child objects DataNucleus starts a larger LDAP search to find the objects (the container DN of the parent class as search base and subtree scope).

Note: Child objects are automatically dependent. If you delete the parent object all child objects are automatically deleted. If you null out the child object reference in the parent object or if you remove the child object from the parents collection, the child object is automatically deleted.

N-1 Unidirectional

This kind of mapping could be used if your LDAP tree has a huge number of child objects and you only work with these child objects.

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                        public class Department {
|                                                            String name;
|-- ou=Sales                                             }
|    |-- cn=Bugs Bunny
|    |-- cn=Daffy Duck                                   public class Employee {
|    |-- ...                                                 String firstName;
|                                                            String lastName;
|-- ou=Engineering                                          String fullName;
|    |-- cn=Speedy Gonzales                                  Department department;
|    |-- ...                                             }
|
|-- ...
```

In the LDAP tree we have departments (Sales and Engineering) and each department holds some associated employees. In our Java classes each **Employee** object knows its **Department** but not vice-versa.

The JDO metadata looks like this:

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="com.example">
        <class name="Department" table="dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
        </class>

        <class name="Employee" table="{department}"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department"/>
        </class>
    </package>
</jdo>
```

The **Department** objects are persisted directly under *dc=example,dc=com*. The **Employee** class has a dynamic DN definition {*department*}. So the DN of the Department instance is used as container for Employee objects.

N-1 (1-N) Bidirectional

If you need a reference from the parent object to the child objects you need to define a bidirectional relationship.

The example LDAP tree and Java classes looks like this:

```
dc=example,dc=com
|
|-- ou=Sales
|   |-- cn=Bugs Bunny
|   |-- cn=Daffy Duck
|   |-- ...
|
|-- ou=Engineering
|   |-- cn=Speedy Gonzales
|   |-- ...
|
|-- ...
```

```
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Department department;
}
```

Now the **Department** class has a Collection containing references to its **Employee**s.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" mapped-by="department"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        </class>

        <class name="Employee" table="{department}"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department"/>
        </class>
    </package>
</jdo>
```

We added a new *employees* field to the Department class that is *mapped-by* the department field of the Employee class.

Please note: When loading the parent object all child object are loaded immediately. For a large number of child entries this may lead to performance and/or memory problems.

1-1 Unidirectional

1-1 unidirectional mapping is very similar to N-1 unidirectional mapping.

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=People
|   |-- cn=Bugs Bunny
|   |   |-- uid=bbunny
|   |
|   |-- cn=Daffy Duck
|   |   |-- uid=dduck
|   |
|   |-- ...
```

```
public class Person {
    String firstName;
    String lastName;
    String fullName;
}

public class Account {
    String uid;
    String password;
    Person person;
}
```

In the LDAP tree we have persons and each person has one account. Each **Account** object knows to which **Person** it belongs to, but not vice-versa.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=People,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>

        <class name="Account" table="{person}"
schema="top,account,simpleSecurityObject">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <field name="uid" primary-key="true column="uid" />
            <field name="password" column="userPasword" />
            <field name="person" />
        </class>
    </package>
</jdo>
```

The **Person** objects are persisted directly under *ou=People,dc=example,dc=com*. The **Account** class has a dynamic DN definition *{person}*. So the DN of the Person instance is used as container for the Account object.

1-1 Bidirectional

If you need a reference from the parent class to the child class you need to define a bidirectional relationship.

The example LDAP tree and Java classes looks like this:

```
dc=example,dc=com                                        public class Person {
|                                                            String firstName;
|-- ou=People                                                String lastName;
|   |                                                        String fullName;
|   |-- cn=Bugs Bunny                                        Account account;
|   |   |-- uid=bbunny                                   }
|   |
|   |-- cn=Daffy Duck                                    public class Account {
|   |   |-- uid=dduck                                        String uid;
|   |                                                        String password;
|   |-- ...                                                  Person person;
                                                         }
```

Now the **Person** class has a reference to its **Account**.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=People,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="account" mapped-by="person" />
        </class>

        <class name="Account" table="{person}"
schema="top,account,simpleSecurityObject">
            <field name="uid" primary-key="true column="uid" />
            <field name="password" column="userPasword" />
            <field name="person" />
        </class>
    </package>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
</jdo>
```

We added a new *account* field to the Person class that is *mapped-by* the person field of the Account class.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.5 Embedded Objects

Embedded Objects

With JDO it is possible to persist field as embedded. This may be useful for LDAP datastores where often many attributes are stored within one entry however logically they describe different objects.

Let's assume we have the following entry in our directory:

```
dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
postalCode: 3578
l: Hollywood
street: Sunset Boulevard
uid: bbunny
userPassword: secret
```

This entry contains multiple type of information: a person, its address and its account data. So we will create the following Java classes:

```
public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Address address;
    Account account;
}

public class Address {
    int zip;
    String city
    String street;
}

public class Account {
    String id;
    String password;
}
```

The JDO metadata to map these objects to one LDAP entry would look like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=Employees,dc=example,dc=com"
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true" column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="account">
                <embedded null-indicator-column="uid">
                    <field name="id" column="uid" />
                    <field name="password" column="userPassword" />
                </embedded>
            </field>
            <field name="address">
                <embedded null-indicator-column="l">
                    <field name="zip" column="postalCode" />
                    <field name="city" column="l" />
                    <field name="street" column="street" />
                </embedded>
            </field>
        </class>
        <class name="Account" embedded-only="true">
            <field name="uid" />
            <field name="password" />
        </class>
        <class name="Address" embedded-only="true">
            <field name="zip" />
            <field name="city" />
            <field name="street" />
        </class>
    </package>
</jdo>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.4.6 Queries

LDAP Queries

Using an LDAP datastore Access Platform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax

It is hoped to provide more options in the future.

When using queries with LDAP there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.5 Excel

Excel Documents

DataNucleus supports persisting objects to Excel documents (using the datanucleus-excel plugin). It makes use of the Apache POI project. Support for Excel is in its early stages and will be enhanced in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

DataNucleus supports the following mode of operation, file-based

The following persistence properties will connect to a local file on your local machine

```
datanucleus.ConnectionURL=excel:file:myfile.xls
```

replacing "myfile.xls" with your filename, which can be absolute or relative

Known Limitations

The following are known limitations of the current implementation

- Relationships between objects are not supported currently and wouldn't be easy to support since Excel doesn't have the concept of a relation
- Optimistic checking of versions is not supported
- Identity generators that operate using the datastore are not supported
- Cannot map inherited classes to the same worksheet

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.5.1 Mapping

Excel Document Mapping

Excel datastore support assumes that objects of a class are persisted to a particular "sheet" of an Excel spreadsheet. Similarly a field of a class is persisted to a particular "column" index of that "sheet". This provides the basis for Excel persistence.

When persisting a Java object to an Excel spreadsheet clearly the user would like some control over what worksheet a class is persisted to, and to what column number a field is persisted. DataNucleus provides extension metadata tags to allow this control. Here's an example, using JDO XML metadata

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true">
            <extension vendor-name="datanucleus" key="sheet" value="People"/>
            <field name="personNum" >
                <extension vendor-name="datanucleus" key="index" value="0"/>
            </field>
            <field name="firstName">
                <extension vendor-name="datanucleus" key="index" value="1"/>
            </field>
            <field name="lastName">
                <extension vendor-name="datanucleus" key="index" value="2"/>
            </field>
        </class>
    </package>
</jdo>
```

So we have two primary extensions in use. The first is **sheet** which is specified for the class as a whole and defines that all objects of class *Person* will be persisted to worksheet called "People". The second is **index** which is specified for a field, and defines that field "personNum" will be persisted as the first column in the worksheet etc.

Here's the same example using JDO Annotations

```
@PersistenceCapable(extensions=@Extension(vendorName="datanucleus", key="sheet",
value="People"))
public class Person
{
    @Extension(vendorName="datanucleus", key="index", value="0")
    long personNum;

    @Extension(vendorName="datanucleus", key="index", value="1")
    String firstName;

    @Extension(vendorName="datanucleus", key="index", value="2")
    String lastName;
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here's the same example using JPA Annotations (with DataNucleus @Extension/@Extensions annotations)

```
@Entity
@Extension(key="sheet", value="People")
public class Person
{
    @Identity
    @Extension(key="index", value="0")
    long personNum;

    @Extension(key="index", value="1")
    String firstName;

    @Extension(key="index", value="2")
    String lastName;
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.5.2 Queries

Excel Queries

Using an Excel datastore Access Platform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax

It is hoped to provide more options in the future.

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.6 XML

XML Datastores

DataNucleus supports persisting objects to XML datastores (using the datanucleus-xml plugin). It makes use of JAXB. Support for XML is in its very early stages and will be enhanced in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

DataNucleus supports the following mode of operation, file-based

The following persistence properties will connect to a local file on your local machine

```
datanucleus.ConnectionURL=xml:file:myfile.xml
```

replacing "myfile.xml" with your filename, which can be absolute or relative

Known Limitations

The following are known limitations of the current implementation

- Datastore identity is not supported
- Optimistic checking of versions is not supported
- Application identity is supported but can only have 1 PK field and must be a String. This is a limitation of JAXB
- Persistent properties are not supported, only persistent fields
- Identity generators that operate using the datastore are not supported

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.6.1 Mapping

XML Datastore Mapping

When persisting a Java object to an XML datastore clearly the user would like some control over the structure of the XML document. Here's an example using JDO XML MetaData

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true" schema="/myproduct/people"
table="person">
            <field name="personNum">
                <extension vendor-name="datanucleus" key="XmlAttribute"
value="true"/>
            </field>
            <field name="firstName" primary-key="true"/> <!-- PK since JAXB requires
String -->
            <field name="lastName"/>
            <field name="bestFriend"/>
        </class>
    </package>
</jdo>
```

Things to note :

- **schema** on class is used to define the "XPath" to the root of the class in XML. You can also use the extension "xpath" to specify the same thing.
- **table** on class is used to define the name of the element for an object of the particular class.
- **column** on field is used to define the name of the element for a field of the particular class.
- **XmlAttribute** : when set to true denotes that this will appear in the XML file as an attribute of the overall element for the object
- When a field is primary-key it will gain a JAXB "XmlID" attribute.
- When a field is a relation to another object (and the field is not embedded) then it will gain a JAXB "XmlIDREF" attribute as a link to the other object.
- **Important : JAXB has a limitation for primary keys** : there can only be a single PK field, and it must be a String!

What is generated with the above is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<myproduct>
    <people>
        <person personNum="1">
            <firstName>Bugs</firstName>
            <lastName>Bunny</lastName>
            <bestFriend>My</bestFriend>
        </person>
    </people>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here's the same example using JDO Annotations

```
@PersistenceCapable(schema="/myproduct/people", table="person")
public class Person
{
    @XmlAttribute
    private long personNum;

    @PrimaryKey
    private String firstName;

    private String lastName;

    private Person bestFiend;

    @XmlElementWrapper(name="phone-numbers")
    @XmlElement(name="phone-number")
    @Element(types=String.class)
    private Map phoneNumbers = new HashMap();

    ...
```

Here's the same example using JPA Annotations (with DataNucleus @Extension/@Extensions annotations)

```
TODO Add this example
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.6.2 Queries

XML Queries

Using an XML "datastore" Access Platform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax

It is hoped to provide more options in the future.

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.7 NeoDatis

Neodatis Datastores

NeoDatis is an object-oriented database for Java and .Net. It is simple and fast and supports various query mechanisms.

DataNucleus supports persisting objects to Neodatis datastores (using the datanucleus-neodatis plugin). Support for Neodatis is in its early stages. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

DataNucleus supports 2 modes of operation of *neodatis* - file-based, and client-server based. In order to do so and to fit in with the JDO/JPA APIs we have defined the following means of connection.

The following persistence properties will connect to a **file-based** Neodatis running on your local machine

```
datanucleus.ConnectionURL=neodatis:file:neodatisdb.odb
```

Replacing "neodatis.odb" by your filename for the datastore, and can be absolute OR relative.

The following persistence properties will connect to **embedded-server-based** NeoDatis running with a local file

```
datanucleus.ConnectionURL=neodatis:server:{my_neodatis_file}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

The filename {my_neodatis_file} can be absolute OR relative.

The following persistence properties will connect as a client to a **TCP/IP NeoDatis Server**

```
datanucleus.ConnectionURL=neodatis:{neodatis_host}:{neodatis_port}/{identifier}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

Neodatis doesn't itself use such URLs so it was necessary to define this DataNucleus-specific way of addressing Neodatis.

Known Limitations

The following are known limitations of the current implementation

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- NeoDatis doesn't have the concept of an "unloaded" field and so when you request an object from the datastore it comes with its graph of objects. Consequently there is no "lazy loading" and the consequent impact that can have on memory utilisation.
- JDOQL for NeoDatis does not support the full range of methods (e.g Collection/Map contains)
- NeoDatis doesn't currently allow access to the version of objects hence optimistic checking of versions is not currently supported

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.7.1 Queries

NeoDatis Queries

Using a NeoDatis datastore DataNucleus allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax
- Native - NeoDatis' own type-safe query language
- Criteria - NeoDatis' own Criteria query language

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.7.2 Native Queries

NeoDatis Native Queries

NeoDatis provides its own "native" query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```
// Find all employees older than 31
Query q = pm.newQuery("Native", new NativeQuery()
    {
        public boolean match(Object e)
        {
            if (!(e instanceof Employee))
            {
                return false;
            }
            return ((Employee)e).getAge() >= 32;
        }
        public Class getObjectType()
        {
            return Employee.class;
        }
    });

List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "NativeQuery".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.7.3 Criteria Queries

NeoDatis Native Queries

NeoDatis provides its own "criteria" query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

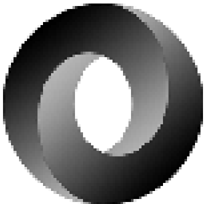
```
// Find all employees older than 31
Query q = pm.newQuery("Criteria", new CriteriaQuery(Employee.class, Where.ge("age",
32)));

List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "CriteriaQuery".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.8 JSON

JSON Datastores



DataNucleus supports persisting objects to JSON datastores (using the datanucleus-json plugin). Support for JSON is in its early stages and will be enhanced in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

The following persistence properties will connect to JSON at a URL

```
datanucleus.ConnectionURL=json:http://www.mydomain.com/somepath/
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.8.1 Mapping

HTTP Mapping

The persistence to JSON datastore is performed via HTTP methods. HTTP response codes are used to validate the success or failure to perform the operations. The JSON datastore must respect the following:

Method	Operation	URL format	HTTP response code
PUT	update objects	/{primary key}	HTTP Code 201 (created), 200 (ok) or 204 (no content)
HEAD	locate objects	/{primary key}	HTTP 404 if the object does not exist
POST	insert objects	/	HTTP Code 201 (created), 200 (ok) or 204 (no content)
GET	fetch objects	/{primary key}	HTTP Code 200 (ok) or 404 if object does not exist
GET	retrieve extent of classes (set of objects)	/	HTTP Code 200 (ok) or 404 if no objects exist
DELETE	delete objects	/{primary key}	HTTP Code 202 (accepted), 200 (ok) or 204 (no content)

Persistent Classes Mapping

Metadata API	Extension Element Attachment	Extension	Description
JDO	/jdo/package/class/extension	url	Defines the location of the resources/objects for the class

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true">
            <extension vendor-name="datanucleus" key="url" value="/Person"/>
        </class>
    </package>
</jdo>
```

In this example, the *url* extension identifies the Person resources/objects as */Person*. The persistence operations will be relative to this path. e.g /Person/{primary key} will be used for PUT (update), GET (fetch) and DELETE (delete) methods.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.8.2 Queries

JSON Queries

Using an JSON datastore Access Platform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax

It is hoped to provide more options in the future.

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.9 ODF

ODF Documents

DataNucleus supports persisting objects to Open Document Format (ODF) documents (using the datanucleus-odf plugin). Such documents can then be used in applications like OpenOffice and KOffice. It makes use of the ODF Toolkit project. Support for ODF is in its early stages and will be enhanced in future releases. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

Datastore Connection

DataNucleus supports the following mode of operation: file-based

The following persistence properties will connect to a local file on your local machine

```
datanucleus.ConnectionURL=odf:file:myfile.ods
```

replacing "myfile.ods" with your filename, which can be absolute or relative

Known Limitations

The following are known limitations of the current implementation

- Persistence of maps is not supported
- Optimistic checking of versions is not supported
- Identity generators that operate using the datastore are not supported
- Cannot map inherited classes to the same worksheet

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.9.1 Mapping

ODF Document Mapping

ODF document support assumes that objects of a class are persisted to a particular "sheet" of an ODF spreadsheet. Similarly a field of a class is persisted to a particular "column" index of that "sheet". This provides the basis for ODF persistence.

When persisting a Java object to an ODF spreadsheet clearly the user would like some control over what worksheet a class is persisted to, and to what column number a field is persisted. DataNucleus provides extension metadata tags to allow this control. Here's an example, using JDO XML metadata

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true" table="People">
            <field name="id" >
                <extension vendor-name="datanucleus" key="index" value="0"/>
            </field>
            <field name="firstName">
                <extension vendor-name="datanucleus" key="index" value="1"/>
            </field>
            <field name="lastName">
                <extension vendor-name="datanucleus" key="index" value="2"/>
            </field>
            ...
        </class>
    </package>
</jdo>
```

Things to note.

- We use **table** attribute of class to define the name of the sheet to use for persistence of this class. We can alternatively use the extension **sheet** as required
- We use the extension **index** to define the column number for the field

Here's the same example using JDO Annotations

```
@PersistenceCapable(table="People")
public class Person
{
    @Extension(vendorName="datanucleus", key="index", value="0")
    long id;

    @Extension(vendorName="datanucleus", key="index", value="1")
    String firstName;

    @Extension(vendorName="datanucleus", key="index", value="2")
    String lastName;

    ...
}
```

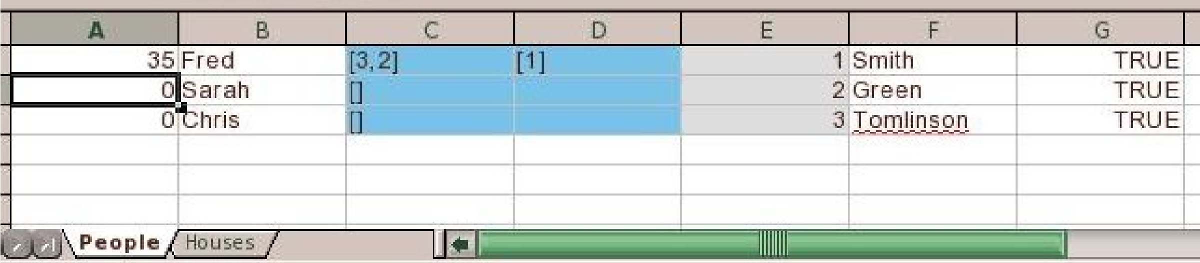
©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here's the same example using JPA Annotations (with DataNucleus @Extension/@Extensions annotations)

```
@Entity
@Table(name="People")
public class Person
{
    @Identity
    @Extension(key="index", value="0")
    long id;

    @Extension(key="index", value="1")
    String firstName;

    @Extension(key="index", value="2")
    String lastName;

    ...
}
```

Relationships

Obviously a spreadsheet cannot store related objects directly since each object is a row of a particular spreadsheet table. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell. See below for an example of the resultant spreadsheet of a class Person that has a 1-1 relation, and a 1-N relation.



A	B	C	D	E	F	G
35	Fred	[3,2]	[1]	1	Smith	TRUE
0	Sarah	[]		2	Green	TRUE
0	Chris	[]		3	Tomlinson	TRUE

In the above example, column 'C' is the 1-N relation, and so elements of the collection are stored as comma-separated object identities. In the same example, column 'D' is the 1-1 relation and the related object is the object identity. When the spreadsheet is read back in again the related object is found.

Table Headers

A typical spreadsheet has many rows of data. It contains no names of columns tying the data back to the input object (field names). DataNucleus allows an extension specified at *class* level called **include-column-headers** (should be set to true). When the table is then created it will include an extra row (the first row) with the column names from the metadata (or field names if no column names were defined). For example

ALL RIGHTS RESERVED

	A	B	C	D	E	F	G	H
1	Age	First Name	Friends	House	Id	Last Name	Single?	
2	35	Fred	[2,3]	[1]	1	Smith	TRUE	
3	0	Sarah	[]		2	Green	TRUE	
4	0	Chris	[]		3	Tomlinson	TRUE	
5								
6								
7								

People Houses

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.9.2 Queries

ODF Queries

Using an ODF document Access Platform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using Java-type syntax

It is hoped to provide more options in the future.

When using queries there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is **datanucleus.query.flushBeforeExecution** and defaults to "false".

To do this on a per query basis for JDOQL/JPQL/SQL using the JDO API you would do

```
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

To do this on a per query basis for JPQL/SQL using the JPA API you would do

```
query.setFlushMode(FlushModeType.AUTO);
```

You can also specify this for all queries using a persistence property **datanucleus.query.flushBeforeExecution** which would then apply to ALL queries for that PMF/EMF.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.10 AppEngine

Google AppEngine - BigTable

Google App Engine (TM) platform provides Java persistence (JDO or JPA) to its BigTable datastore using the datanucleus-appengine plugin. This plugin was developed by Google with the assistance of the DataNucleus project and its capabilities are documented on their site.

Datastore Connection

The following persistence properties will connect to AppEngine

```
datanucleus.ConnectionURL=appengine
```

Reviews

Google AppEngine (TM) for Java provides developers with a simple environment in which to develop and deploy web based applications. There have been many blogs about this in recent days. Below are some

- GWT and JDO on the Java App Engine
- 1st look at App Engine using JDO persistence capable classes over GWT RPC
- Google AppEngine for java, first impressions
- Google AppEngine Java Test Ride
- Google App Engine adds Java support (Review)
- Googles Groovy AppEngine
- Google AppEngine for Java with Rich Ruby clients
- Google AppEngine limitations for Java (and how to overcome them)
- Google AppEngine and GWT now a marriage made in heaven
- Re-thinking Object-Relational Mapping in a Distributed Key-Store world

If you find others and think others would benefit from them please notify us via the forum

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.1 Core

DataNucleus Core

datanucleus-core is the base persistence implementation of DataNucleus. All other DataNucleus projects build on top of this, and so it is the pre-requisite for any DataNucleus-enabled application. Includes JDO persistence.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-core is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-core is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
JDO2	2.3+	Apache JDO2 API.	Yes
log4j	1.2.x+	Apache Log4J Logging framework.	No. Use it or JDK1.4 logging

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.2 Enhancer

DataNucleus Enhancer

datanucleus-enhancer provides bytecode enhancement of classes that will be involved in the persistence process. It adds in the necessary methods required to intercept changes in the objects hence providing the speed benefits of DataNucleus.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-enhancer is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-enhancer is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
JDO2	2.3+	Apache JDO2 API.	Yes
ASM	3.0+	ASM byte code manipulation framework. This is used to byte code enhance the classes.	Yes
log4j	1.2.x+	Apache Log4J Logging framework.	No. Use it or JDK1.4 logging

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.3 JPA

DataNucleus JPA

datanucleus-jpa provides support for the DataNucleus JPA implementation.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-jpa is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-jpa is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1+	Provides JDO, logging, localisation, and other framework facilities for DataNucleus	Yes
JPA1	1.0+	Sun Java Persistence API (JPA). Alternatively use Geronimo JPA jar (Apache license)	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.4 Cache

DataNucleus Cache

datanucleus-cache provides Level2 caching capabilities. It supports use of EHCache, Oracle Coherence, OSCache, and SwarmCache.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-cache is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-cache is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
ehcache	1.1	EHCache caching	Yes if using EHCache
Oracle Coherence		Oracle Coherence caching	Yes, if using Coherence
oscache	2.1	OSCache caching	Yes, is using OSCache
swarmcache	1.0RC2	SwarmCache caching	Yes, if using SwarmCache

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.5 REST

DataNucleus REST

datanucleus-rest provides support to RESTful access to the datastore.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.0.0
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-rest is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven2 Nightly Repository

Dependencies

datanucleus-rest is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1+	Provides JDO, logging, localisation, and other framework facilities for DataNucleus	Yes
Flexjson	1.7+	Flexjson	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.6 Management

DataNucleus Management (JMX)

datanucleus-management provides support for JMX management of features, using the MX4J JMX server or the JRE1.5+ JMX server.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-management is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-management is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
mx4j	3.0+	MX4J	Yes, if using MX4J
mx4j-tools	3.0+	MX4J Tools	Yes, if using MX4J
javax.management	1.2+	JMX	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.7 JDO > Connector

DataNucleus JDO JCA

datanucleus-jca provides support using DataNucleus JDO within J2EE environments with managed connections.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-jca is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-jca is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.8 Store > RDBMS

DataNucleus RDBMS

datanucleus-rdbms provides persistence of Java objects to RDBMS datastores. It builds on top of the basic persistence provided by *datanucleus-core*

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-rdbms is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-rdbms is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.8.1 ConnectionPool

DataNucleus ConnectionPool

datanucleus-connectionpool provides datasource pooling when used with RDBMS datastores. It supports use of Apache DBCP, C3P0, and Proxool.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-connectionpool is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-connectionpool is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.0+	Persistence to RDBMS datastores	Yes
commons-dbcp	1.1+	Apache Commons DBCP. Connection Pooling	Yes, if using DBCP
commons-pool	1.1+	Apache Commons Pool.	Yes, if using DBCP
commons-collections	3.0+	Apache Commons Collections	Yes, if using DBCP
c3p0	0.9.0+	C3P0 Connection Pooling	Yes, if using C3P0
proxool		Proxool Connection Pooling	Yes, if using Proxool

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.8.2 Spatial

DataNucleus Spatial

datanucleus-spatial provides support for persistence of many geometric/spatial datatypes. It builds on top of the *datanucleus-core* persistence capabilities, and is used in conjunction with the *datanucleus-rdbms* plugin.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.1.0
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-spatial is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-spatial is dependent on the following packages of software and this differs for each mapping scenario. Please refer to the Spatial Guide for more information about these scenarios.

jgeom2mysql

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.0+	Persistence to RDBMS datastores	Yes
sdoapi	-	Oracle SDO API, contains the JGeometry class.	No
mysql-connector-java	-	MySQL JDBC driver.	No

jgeom2oracle

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
sdoapi	-	Oracle SDO API, contains the JGeometry class .	No
classes12/ojdbc14	-	Oracle JDBC, use classes12 for Java 1.3 and ojdbc14 for all later versions.	No

jts2mysql

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
JTS	1.6+	Java Topology Suite (JTS)	No
mysql-connector-java	-	MySQL JDBC driver.	No

jts2oracle

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
JTS	1.6+	Java Topology Suite (JTS)	No
JTSIO	1.6+	IO package for JTS	No
classes12/ojdbc14	-	Oracle JDBC, use classes12 for Java 1.3 and ojdbc14 for all later versions.	No

jts2postgis

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Package	Version	Description	Required
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
JTS	1.6+	Java Topology Suite (JTS)	No
PostGIS-JTS	1.1.6+	PostGIS JDBC driver extension for JTS types. For correct transformation of 3d-geometries you need a version newer than 1.2.0 (at the moment only available as source from SVN)	No
PostgreSQL	-	PostgreSQL-JDBC driver	No

pg2mysql

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
PostGIS	1.1.6+	PostGIS JDBC driver and PostGIS geometries	No
mysql-connector-java	-	MySQL JDBC driver.	No

pg2postgis

Package	Version	Description	Required
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.1+	Persistence to RDBMS datastores	Yes
PostGIS	1.1.6+	PostGIS JDBC driver and PostGIS geometries	No
PostgreSQL	-	PostgreSQL-JDBC driver	No

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.8.3 XMLTypeOracle

DataNucleus XMLTypeOracle

datanucleus-xmltypeoracle provides support for persisting the Oracle type "XMLType" to Oracle RDBMS datastores, when using the *datanucleus-rdbms* plugin.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-xmltypeoracle is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-xmltypeoracle is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0.x	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-rdbms	1.0+	Persistence to RDBMS datastores	Yes
Oracle XDB		Oracle XDB	Yes
Oracle XMLParServ		Oracle XMLParServ	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.9 Store > DB4O

DataNucleus DB4O

datanucleus-db4o provides persistence of Java objects to DB4O datastores. It builds on top of the basic persistence provided by *datanucleus-core*

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-db4o is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-db4o is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
sql4o	1.0+	SQL querying of DB4O	Yes, if using SQL
DB4O	7.0+	DB4O Datastore	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.9.1 SQL4O

DataNucleus sql4o

datanucleus-sql4o provides support for SQL querying of db4o datastores. It can be used outside of a DataNucleus environment, and originated as a GoogleCode project. Version 1.0 is for use with JDK1.5 (JDBC 2/3). Version 1.1 is for use with JDK1.6+ (JDBC4).

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.0
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1, 1.0
- Source : latest

Download

datanucleus-sql4o is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-sql4o is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
db4o	5.5+	DB4O object datastore	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.10 Store > LDAP

DataNucleus LDAP

datanucleus-ldap provides persistence of Java objects to LDAP. It builds on top of the basic persistence provided by *datanucleus-core*.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.1.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.1
- Source : latest

Download

datanucleus-ldap is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-ldap is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
Apache Directory Shared	0.9.13	To use native LDAP queries you need a subset of Apache Directory Shared: • Apache Directory Protocol Ldap Shared 0.9.13 • Apache Directory Protocol Ldap Shared Constants 0.9.13 • Apache Directory ASN.1 Shared 0.9.13 • SLF4J API Module 1.3.1 • SLF4J LOG4J-12 Binding 1.3.1	Yes, if using native LDAP queries

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.11 Store > Excel

DataNucleus Excel

datanucleus-excel provides persistence of Java objects to Excel documents. It builds on top of the basic persistence provided by *datanucleus-core*.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.5
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-excel is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-excel is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+x	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
Apache POI	3.2	Apache POI, providing Java API access to Microsoft file formats	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.12 Store > XML

DataNucleus XML

datanucleus-xml provides persistence of Java objects to XML datastores. It builds on top of the basic persistence provided by *datanucleus-core*

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.0.6
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-xml is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-xml is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
jaxb-api	2.1	JAXB API	Yes
jaxb-impl	2.x	JAXB Implementation	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.13 Store > Neodatis

DataNucleus Neodatis

datanucleus-neodatis provides persistence of Java objects to Neodatis datastores. It builds on top of the basic persistence provided by *datanucleus-core*

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.6
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-neodatis is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-neodatis is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
Neodatis	1.9	Neodatis Datastore	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.14 Store > JSON

DataNucleus JSON

datanucleus-json provides persistence of Java objects to JSON objects (REST). It builds on top of the basic persistence provided by *datanucleus-core*.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.0.3
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-json is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-json is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.15 Store > ODF

DataNucleus ODF (OpenOffice)

datanucleus-odf provides persistence of Java objects to ODF (OpenOffice) documents. It builds on top of the basic persistence provided by *datanucleus-core*.

- Compile Requirement : JDK1.5+
- Runtime Requirement : JDK1.5+
- Latest version : 1.0.1
- Roadmap : Via JIRA
- License : Apache 2
- Javadocs : 1.0
- Source : latest

Download

datanucleus-odf is downloadable as following

- **Nightly builds** from Maven1 Nightly Repository

Dependencies

datanucleus-odf is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0+x	Provides logging, localisation, and other framework facilities for DataNucleus.	Yes
ODFDOM (ODF Toolkit)	0.6.16	ODF Toolkit	Yes
Apache Xerces	2.8.1	Xerces XML parser, required by ODF Toolkit	Yes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.16 Maven1 Plugin

DataNucleus Plugin

DataNucleus Maven1 Plugin provides a plugin for use of DataNucleus with Maven 1.x.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.1.0
- Roadmap : Via JIRA
- License : Apache 2
- Source : latest

Download

DataNucleus Maven1 Plugin is downloadable as following

- **Releases** from SourceForge
- **Releases** from Maven1 Repository
- **Nightly builds** from Maven1 Nightly Repository

Dependencies

DataNucleus Maven1 Plugin is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.0.x	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-enhancer	1.0.x	Bytecode enhancement of classes to be persisted	Yes if using enhancement functionality
datanucleus-rdbms	1.0.x	Persistence to RDBMS datastore	Yes if using SchemaTool functionality

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.17 Maven2 Plugin

DataNucleus Maven2 Plugin

DataNucleus Maven2 Plugin provides a plugin for use of DataNucleus with Maven 2.x.

- Compile Requirement : JDK1.3+
- Runtime Requirement : JDK1.3+
- Latest version : 1.1.2
- Roadmap : Via JIRA
- License : Apache 2
- Source : latest

Download

DataNucleus Maven2 Plugin is downloadable as following

- **Releases** from SourceForge
- **Releases** from JPOX Maven2 Repository

Dependencies

DataNucleus Maven2 Plugin is dependent on the following packages of software. Click on the name to go to the home page for that software to download it.

Package	Version	Description	Required?
datanucleus-core	1.1.x	Provides logging, localisation, and other framework facilities for DataNucleus	Yes
datanucleus-enhancer	1.1.x	Bytecode enhancement of classes to be persisted	Yes if using enhancement functionality
datanucleus-rdbms	1.1.x	Persistence to RDBMS datastore	Yes if using SchemaTool functionality
datanucleus-jpa	1.1.x	JPA handling	Yes if using JPA

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

16.1 Plugins

Plugin Extensions

DataNucleus is an extensible persistence tool. The DataNucleus Core persistence engine allows the user to plug in many user extensions which will contribute to augment the numerous persistence aspects faced by the Object/Datastore mapping. Plugins are loaded by a plugin manager which uses a registry mechanism, inspecting jars in the CLASSPATH.

These plugins are defined in OSGi format. This format relies on there being a set of *extension-point*s where DataNucleus can be extended, and then adding *extension*s for each of these points. The format means that with very little effort any user can provide their own DataNucleus plugins and have them available very quickly. In short, the three steps necessary for creating a DataNucleus plugin are

1. Review the DataNucleus interface that you will need to implement to generate the plugin, and implement it
2. Create a file *plugin.xml* at the top level of your JAR defining the plugin details (see below).
3. Update the META-INF/MANIFEST.MF file contained in the jar so that it includes necessary information for OSGi.

A minimum META-INF/MANIFEST.MF for a plugin jar should look like this

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: myplugin
Bundle-SymbolicName: org.datanucleus.myplugin
Bundle-Version: 1.0.0
Bundle-Vendor: My Company
```

If you write a DataNucleus plugin and you either want it to be included in the DataNucleus distribution, or want it to be listed here then please contact us via the DataNucleus Forum

Plugins : Non-managed environment

Non managed environment is a runtime environment where DataNucleus runs and plug-ins are not managed by a container. In this environment the plug-in discovery and lifecycle is managed by DataNucleus.

There is a 1 to N instance relationship from DataNucleus to a plug-in per PMF. More exactly, if only one PMF exists, there is only one Plug-in instance for a Connection Pool Plug-in, and if "N" PMF exist, there are "N" Plug-in instances for a Connection Pool Plug-in.

* draw diagram

Standard Java runtimes and J2EE containers are considered non managed environments. In non managed environments there is no lifecycle itself of plug-ins. Extensions implemented by plug-ins are instantiated on demand and terminated on PMF closing, PM closing or in another form depending in what the extension is used for.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Plugins : Managed environment

Managed environment is a runtime environment where DataNucleus plug-ins are managed by a container. The discovery, registry and lifecycle of plug-ins are controlled by the container. There is no plug-in instance relationship from DataNucleus to a plug-in regarding PMF instances. In managed environments, there is only one plug-in instance for one or "N" PMFs. Again, this is managed by the container.

* draw object diagram

DataNucleus supports OSGi containers as managed environment. In OSGi managed environments plug-in lifecycle is determined by OSGi specification. Once activated, a plug-in is only stopped when the OSGi container finishes its execution, or the plug-in is stopped by an OSGi command.

Plug-ins : Extension Points and Extensions

DataNucleus has mechanism that allows extending its persistence engine while not coupling the DataNucleus engine to the extensions. This mechanism is known as "Extension Points", which are well defined interfaces that allows the extension of certain aspects of DataNucleus in a dynamic and consistent model. The "Extensions" are the implementation of Extension Points that are registered in a PluginRegistry and used by DataNucleus Core or other DataNucleus Plug-ins. A plugin owns an Extension. The behaviour is defined below :-

- **Lifecycle** : Each extension is created by a segment of code during the runtime execution, and destroyed/released whenever they are no longer needed. This have no influence with the plug-in lifecycle.
- **Manageability** : In non managed environments, the plug-ins are managed by DataNucleus and maintained with a composition relation to the PMF instance. This allows a plug-in "instance" per PMF. If multiple PMFs are created multiple extensions for an extension point are instantiated. In managed environments, more precisely in OSGi containers, the plug-ins are managed by the OSGi framework. Each plug-in will mostly be a singleton inside the OSGi container.
- **Registration** : In non managed environments all plugins are registered using an instance of JDOClassLoaderResolver (so using the current ClassLoader of the PMF and the current thread). This means that the /plugin.xml and /META-INF/MANIFEST.MF files must be accessible to the classloader. In managed environment this is handled by the container.
- **ClassLoading** : The classloading in non managed environments is usually made of one single ClassLoader, while in managed environments each plug-in has it's own ClassLoader.
- **Configuration** : Some Extensions needs to retrieve a configuration that was set in the PMF. This means that Plug-ins should not hold singleton / static configurations if they want to serve to multiple PMFs at the same time.
- **Constructors/Methods** : In order of having consistent and avoid changes to extension-point interfaces, the Extension Constructors or Methods (either one) should have receive a PMFContext instance as argument. If by the time the Extension Point is designed clearly there is usage for a PMFContext, then the Extension-Point does not need to take the PMFContext as argument, but keep in mind that a 3rd Extension may need one due to different reasons.
- **Instantiation** : Inside the DataNucleus Core, regardless if the runtime is OSGi managed or non managed, extension instances are created per PMF. DataNucleus Extensions should always be created

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

through a PluginManager, regardless if the managed environment would allow you to instantiate using their own interfaces. This allows DataNucleus and its Plug-ins to run in non managed environments.

Plugin Properties

Plugins can make use of existing persistence properties defined by "core" etc. They can also add on their own persistence properties. They define these in the *plugin.xml*. Let's take an example

```
    <!-- PERSISTENCE PROPERTIES -->
    <extension point="org.datanucleus.persistence_properties">
        <persistence-property name="datanucleus.rdbms.query.fetchDirection"
value="forward"
            validator="org.datanucleus.store.rdbms.RDBMSPropertyValidator"/>
    </extension>
```

So here we have a property defined in the "store.rdbms" plugin which defines a persistence property *org.datanucleus.rdbms.query.fetchDirection* with a default value of *forward* and with values validated by the class *org.datanucleus.store.rdbms.RDBMSPropertyValidator*. This means that when we instantiate a PMF/EMF we automatically get that property defined with value "forward", and if we provide it ourselves we override this default. We do the same with any properties we want in our own plugins.

In order to avoid conflicts between different property names that can be set in the PMF and to have a consistent naming schema for properties, the following recommendations should be applied.

Naming the persistence property name should be prefixed by the plug-in id. Example:

```
Plugin: org.datanucleus.myplugin1
Property: myprop1
```

The persistence property should look like *datanucleus.myplugin1.myprop1*.

If an extension point defines a new persistence configuration, the property name should be prefixed by the extension-point id. Example:

```
Extension-Point id: org.datanucleus.myplugin2.myextensionpoint2
Property: myprop2
```

The persistence property should look like *datanucleus.myplugin2.myextensionpoint2.myprop2*.

Another form of persistence configuration could happen if multiple Extension-Points uses the same information. In this case an abstraction of the plug-in id and extension point id could be used, as the following example. However, make sure to use this naming schema only when absolutelly necessary.

```
Extension-Point id: org.datanucleus.myplugin3.myextensionpoint3
Extension-Point id: org.datanucleus.yourplugin3.yourextensionpoint4
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Extension-Point id: org.datanucleus.theirplugin4.theirextensionpoint5
Property: myprop345
```

The persistence property should look like *datanucleus.somepluginXXXX.someextensionpointYYYY.myprop345*. Alternativelly, the PMF property could also look like *datanucleus.myplugin3.myextensionpoint3.myprop345*

Integrating with 3rd party products

It's a goal of the DataNucleus project to seamless integrate to all major 3rd party products used by the DataNucleus community. In addition to the plugins above it also provides plugins for the Eclipse IDE. All plugins can be downloaded here.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.1 Java Types

Plugins : Java Type

DataNucleus provides capabilities for persistence of particular Java types. Some types are by default persistent, some are by default in the default "fetch-group". Similarly some are second class mutable, and hence have their operations intercepted. An extension-point is available to define other Java types in this way. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.java_type*.

The attributes that you can set for each Java type are

- **dfg** - whether this type is by default in the default-fetch-group (true/false)
- **persistent** - whether this type is, by default, persistent (true/false)
- **embedded** - whether this type is, by default, embedded (true/false)
- **wrapper-type** - class name of the SCO wrapper (if it needs a wrapper)
- **wrapper-type-backed** - class name of a SCO wrapper (with backing store)
- **java-version** - version of JDK that this applies to (if specific only)
- **java-version-restricted** - whether this is restricted to a particular version of JDK
- **string-converter** - name of a class that can convert this type to/from String

All of these are optional, and you should define what is required for your type.

wrapper-type

As we've mentioned above, if a java type is considered *second class mutable* then it needs to have any mutating operations intercepted. The reason for this is that DataNucleus needs to be aware when the type has changed value internally. To give an example of such a type and how you would define support for intercepting these mutating operations lets use *java.util.Date*. We need to write a *wrapper* class. This has to be castable to the same type as the Java type it is representing (so inherited from it). So we extend "java.util.Date", and we need to implement the interface *org.datanucleus.sco.SCO*

```
package org.mydomain;

import java.io.ObjectStreamException;
import javax.jdo.JDOHelper;
import javax.jdo.spi.PersistenceCapable;
import org.datanucleus.StateManager;

public class MyDateWrapper extends java.util.Date implements SCO
{
    private transient StateManager ownerSM;
    private transient Object owner;
    private transient String fieldName;

    public MyDateWrapper(StateManager ownerSM, String fieldName)
    {
        super();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        if (ownerSM != null)
        {
            this.ownerSM = ownerSM;
            this.owner = ownerSM.getObject();
        }
        this.fieldName = fieldName;
    }

    public void initialise()
    {
    }

    /** Method to initialise the SCO from an existing value. */
    public void initialise(Object o, boolean forInsert, boolean forUpdate)
    {
        super.setTime(((java.util.Date)o).getTime());
    }

    /** Wrapper for the setTime() method. Mark the object as "dirty" */
    public void setTime(long time)
    {
        super.setTime(time);
        makeDirty();
    }

    /** Wrapper for the setYear() deprecated method. Mark the object as "dirty" */
    public void setYear(int year)
    {
        super.setYear(year);
        makeDirty();
    }

    /** Wrapper for the setMonth() deprecated method. Mark the object as "dirty" */
    public void setMonth(int month)
    {
        super.setMonth(month);
        makeDirty();
    }

    /** Wrapper for the setDates() deprecated method. Mark the object as "dirty" */
    public void setDate(int date)
    {
        super.setDate(date);
        makeDirty();
    }

    /** Wrapper for the setHours() deprecated method. Mark the object as "dirty" */
    public void setHours(int hours)
    {
        super.setHours(hours);
        makeDirty();
    }

    /** Wrapper for the setMinutes() deprecated method. Mark the object as "dirty"
*/
    public void setMinutes(int minutes)
    {
        super.setMinutes(minutes);
        makeDirty();
    }

    /** Wrapper for the setSeconds() deprecated method. Mark the object as "dirty"
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
*/
    public void setSeconds(int seconds)
    {
        super.setSeconds(seconds);
        makeDirty();
    }

    /** Accessor for the unwrapped value that we are wrapping. */
    public Object getValue()
    {
        return new java.util.Date(getTime());
    }

    public Object clone()
    {
        Object obj = super.clone();
        ((Date)obj).unsetOwner();
        return obj;
    }

    public void unsetOwner()
    {
        owner = null;
        ownerSM = null;
        fieldName = null;
    }

    public Object getOwner()
    {
        return owner;
    }

    public String getFieldName()
    {
        return this.fieldName;
    }

    public void makeDirty()
    {
        if (ownerSM != null)
        {
            ownerSM.getObjectManager().getApiAdapter().makeFieldDirty(owner,
fieldName);
        }
    }

    public Object detachCopy(FetchPlanState state)
    {
        return new java.util.Date(getTime());
    }

    public void attachCopy(Object value)
    {
        long oldValue = getTime();
        initialise(value, false, true);

        // Check if the field has changed, and set the owner field as dirty if
necessary
        long newValue = ((java.util.Date)value).getTime();
        if (oldValue != newValue)
        {
            makeDirty();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }
    }

    /**
     * Handling for serialising our object.
     */
    protected Object writeReplace() throws ObjectStreamException
    {
        return new java.util.Date(this.getTime());
    }
}
```

So we simply intercept the mutators and mark the object as dirty in its StateManager.

string-converter

If your Java type is not able to be persisted natively by a datastore (for example a URL is not storable as a URL type in an Excel spreadsheet) then you could provide a way of storing it as a String since all datastores allow Strings to be stored. Let's take an example. We have a java type *java.net.URL*. We want to provide a **string-converter**. We need to implement *org.datanucleus.store.types.ObjectStringConverter* So we define our converter like this

```
public class URLStringConverter implements ObjectStringConverter
{
    public Object toObject(String str)
    {
        if (str == null)
        {
            return null;
        }

        URL url = null;
        try
        {
            url = new java.net.URL(str.trim());
        }
        catch (MalformedURLException mue)
        {
            throw new NucleusDataStoreException(
                "Error converting String \"" + str + "\" to URL", mue);
        }
        return url;
    }

    public String toString(Object obj)
    {
        String str;
        if (obj instanceof URL)
        {
            str = ((URL)obj).toString();
        }
        else
        {
            str = (String)obj;
        }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        return str;
    }
}
```

So as simple as it could possibly be. We implement method *toObject(String)* and *toString(Object)*. The vast majority of java types could provide this type of conversion as a fallback if the datastore doesn't handle their type natively.

Plugin Specification

To define the persistence characteristics of a Java type you need to add entries to a *plugin.xml* file at the root of the CLASSPATH. The file *plugin.xml* will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mystore" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.java_type">
        <java-type name="java.util.Date" wrapper-type="mydomain.MyDateWrapper"
persistent="true" dfg="true"/>
    </extension>
</plugin>
```

Note that you also require a MANIFEST.MF file as per the Plugins Guide.

Obviously all standard types (such as *java.util.Date*) already have their values defined by DataNucleus itself typically in *datanucleus-core*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.2 Store Manager

Plugins : Store Manager

DataNucleus provides support for persisting objects to particular datastores. It provides this capability via a "Store Manager". It provides a Store Manager plugin for RDBMS datastores. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_manager*.

Plugin extension-point	Key	Description	Location
org.datanucleus.store_manager	rdbms	Store Manager for RDBMS datastores	datanucleus-rdbms
org.datanucleus.store_manager	db4o	Store Manager for DB4O datastore	datanucleus-db4o
org.datanucleus.store_manager	excel	Store Manager for Excel documents	datanucleus-excel
org.datanucleus.store_manager	ldap	Store Manager for LDAP datastores	datanucleus-ldap
org.datanucleus.store_manager	xml	Store Manager for XML datastores	datanucleus-xml
org.datanucleus.store_manager	neodatis	Store Manager for NeoDatis datastores	datanucleus-neodatis
org.datanucleus.store_manager	json	Store Manager for JSON datastores	datanucleus-json
org.datanucleus.store_manager	odf	Store Manager for ODF datastores	datanucleus-odf

Interface

If you want to implement support for another datastore you can achieve it by implementating the StoreManager interface.

Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file *plugin.xml* in your JAR at the root of the CLASSPATH. The file *plugin.xml* will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mystore" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.store_manager">
        <store-manager class-name="mydomain.MyStoreManager" url-key="mykey"
key="mykey"/>
    </extension>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
</plugin>
```

Plugin Usage

The only thing remaining is to use your StoreManager. To do this you simply define your ConnectionURL to start with the *mykey* defined in the plugin spec. This will select your store manager based on that.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.3 AutoStart Mechanisms

Plugins : AutoStart Mechanism

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the mechanism for starting with knowledge of previously persisted classes. DataNucleus provides 3 "auto-start" mechanisms, but also allows you to plugin your own variant.

DataNucleus can discover the classes that it is managing at runtime, or you can use an "autostart" mechanism to inform DataNucleus of what classes it will be managing. DataNucleus provides a selection of plugins for autostart mechanism. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.autostart*.

Plugin extension-point	Key	Description	Location
org.datanucleus.autostart	classes	AutoStart mechanism specifying the list of classes to be managed	datanucleus-core
org.datanucleus.autostart	xml	AutoStart mechanism using an XML file to store the managed classes	datanucleus-core
org.datanucleus.autostart	schematable	AutoStart mechanism using a table in the RDBMS datastore to store the managed classes	datanucleus-rdbms

Interface

Any auto-start mechanism plugin will need to implement *org.datanucleus.store.AutoStartMechanism*. So you need to implement the following interface

```
package org.datanucleus.store;

public interface AutoStartMechanism
{
    /** mechanism is disabled if None **/
    public static final String NONE = "None";

    /** mechanism is in Quiet mode **/
    public static final String MODE_QUIET = "Quiet";

    /** mechanism is in Checked mode **/
    public static final String MODE_CHECKED = "Checked";

    /** mechanism is in Ignored mode **/
    public static final String MODE_IGNORED = "Ignored";

    /**
     * Accessor for the mode of operation.
     * @return The mode of operation
     **/
    String getMode();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Mutator for the mode of operation.
     * @param mode The mode of operation
     **/
    void setMode(String mode);

    /**
     * Accessor for the data for the classes that are currently auto started.
     * @return Collection of {@link StoreData} elements
     * @throws DatastoreInitialisationException
     **/
    Collection getAllClassData() throws DatastoreInitialisationException;

    /**
     * Starts a transaction for writing (add/delete) classes to the auto start
mechanism.
     */
    void open();

    /**
     * Closes a transaction for writing (add/delete) classes to the auto start
mechanism.
     */
    void close();

    /**
     * Whether it's open for writing (add/delete) classes to the auto start
mechanism.
     * @return whether this is open for writing
     */
    public boolean isOpen();

    /**
     * Method to add a class/field (with its data) to the currently-supported list.
     * @param data The data for the class.
     **/
    void addClass(StoreData data);

    /**
     * Method to delete a class/field that is currently listed as supported in
     * the internal storage.
     * It does not drop the schema of the DatastoreClass
     * neither the contents of it. It only removes the class from the
     * AutoStart mechanism.
     * TODO Rename this method to allow for deleting fields
     * @param name The name of the class/field
     **/
    void deleteClass(String name);

    /**
     * Method to delete all classes that are currently listed as supported in
     * the internal storage. It does not drop the schema of the DatastoreClass
     * neither the contents of it. It only removes the classes from the
     * AutoStart mechanism.
     **/
    void deleteAllClasses();

    /**
     * Utility to return a description of the storage for this mechanism.
     * @return The storage description.
     **/
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    String getStorageDescription();
}
```

You can extend *org.datanucleus.store.AbstractAutoStartMechanism*

Implementation

So lets assume that you want to create your own auto-starter **MyAutoStarter**.

```
package mydomain;

import org.datanucleus.store.AutoStartMechanism;
import org.datanucleus.store.AbstractAutoStartMechanism;
import org.datanucleus.store.StoreManager;
import org.datanucleus.ClassLoaderResolver;

public class MyAutoStarter extends AbstractAutoStartMechanism
{
    public MyAutoStarter(StoreManager storeMgr, ClassLoaderResolver clr)
    {
        super();
    }

    ... (implement the required methods)
}
```

Plugin Specification

When we have defined our "AutoStartMechanism" we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.autostart">
        <autostart name="myStarter" class-name="mydomain.MyAutoStarter"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your new *AutoStartMechanism* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property **org.datanucleus.autoStartMechanism** to *myStarter* (the name you specified in the plugin.xml file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.4 ClassLoader Resolvers

Plugins : ClassLoader Resolvers

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the class-loading resolution. DataNucleus provides its own internal class-loader resolver that matches the requirements of the JDO2 specification, but also allows you to plugin your own class loading policy. This class loader resolver is used for runtime operation. The plugin mechanism always operates using an instance of JDOClassLoaderResolver since we need a class loading process to discover the plugins in the first place.

DataNucleus has to resolve classes at runtime. The JDO2 specification defines a strict process for resolving classes using specific class loaders. DataNucleus provides a plugin for this JDO2 process. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.classloader_resolver*.

Plugin extension-point	Key	Description	Location
org.datanucleus.classloader_resolver	jdo	ClassLoaderResolver using JDO2 definition	datanucleus-core

JDO2 Implementation

At runtime, DataNucleus will need to load the classes being persisted. To do this it needs to utilise a particular class loading mechanism. The JDO2 specification defines how class-loading is to operate in a JDO implementation and this is what DataNucleus uses by default. In brief the JDO2 class-loading mechanism utilises 3 class loaders

- When creating the PersistenceManagerFactory you can specify a class loader. This is used first if specified
- The second class loader to try is the class loader for the current thread.
- The third class loader to try is the class loader for the PMF context.

If a class cannot be loaded using these three loaders then an exception is typically thrown depending on the operation.

Interface

Any identifier factory plugin will need to implement *org.datanucleus.ClassLoaderResolver*. So you need to implement the following interface

```
package org.datanucleus;
...
public interface ClassLoaderResolver
{
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Class loading method, allowing specification of a primary loader.
     * This method does not initialize the class
     * @param name Name of the Class to be loaded
     * @param primary the primary ClassLoader to use (or null)
     * @return The Class given the name, using the specified ClassLoader
     * @throws ClassNotResolvedException if the class can't be found in the
classpath
     */
    public Class classForName(String name, ClassLoader primary);

    /**
     * Class loading method, allowing specification of a primary loader
     * and whether the class should be initialised or not.
     * @param name Name of the Class to be loaded
     * @param primary the primary ClassLoader to use (or null)
     * @param initialize whether to initialize the class or not.
     * @return The Class given the name, using the specified ClassLoader
     * @throws ClassNotResolvedException if the class can't be found in the
classpath
     */
    public Class classForName(String name, ClassLoader primary, boolean initialize);

    /**
     * Class loading method. This method does not initialize the class
     * @param name Name of the Class to be loaded
     * @return The Class given the name, using the specified ClassLoader
     */
    public Class classForName(String name);

    /**
     * Class loading method, allowing for initialisation of the class.
     * @param name Name of the Class to be loaded
     * @param initialize whether to initialize the class or not.
     * @return The Class given the name, using the specified ClassLoader
     */
    public Class classForName(String name, boolean initialize);

    /**
     * Method to test whether the type represented by the specified class_2
     * parameter can be converted to the type represented by class_name_1 parameter.
     * @param class_name_1 Class name
     * @param class_2 Class to compare against
     * @return Whether they are assignable
     */
    public boolean isAssignableFrom(String class_name_1, Class class_2);

    /**
     * Method to test whether the type represented by the specified class_name_2
     * parameter can be converted to the type represented by class_1 parameter.
     * @param class_1 First class
     * @param class_name_2 Class name to compare against
     * @return Whether they are assignable
     */
    public boolean isAssignableFrom(Class class_1, String class_name_2);

    /**
     * Method to test whether the type represented by the specified class_name_2
     * parameter can be converted to the type represented by class_name_1 parameter.
     * @param class_name_1 Class name
     * @param class_name_2 Class name to compare against
     * @return Whether they are assignable
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     */
    public boolean isAssignableFrom(String class_name_1, String class_name_2);

    /**
     * ClassLoader registered to load classes created at runtime. One ClassLoader
can
     * be registered, and if one ClassLoader is already registered, the registered
ClassLoader
     * is replaced by <code>loader</code>.
     * @param loader The ClassLoader in which classes are defined
     */
    public void registerClassLoader(ClassLoader loader);

    /**
     * ClassLoader registered by users to load classes. One ClassLoader can
     * be registered, and if one ClassLoader is already registered, the registered
ClassLoader
     * is replaced by <code>loader</code>.
     * @param loader The ClassLoader in which classes are loaded
     */
    public void registerUserClassLoader(ClassLoader loader);

    /**
     * Finds all the resources with the given name.
     * @param resourceName the resource name. If <code>resourceName</code> starts
with "/",
     *           remove it before searching.
     * @param primary the primary ClassLoader to use (or null)
     * @return An enumeration of URL objects for the resource. If no resources could
be found,
     *           the enumeration will be empty.
     * Resources that the class loader doesn't have access to will not be in the
enumeration.
     * @throws IOException If I/O errors occur
     * @see ClassLoader#getResources(java.lang.String)
     */
    public Enumeration getResources(String resourceName, ClassLoader primary) throws
IOException;

    /**
     * Finds the resource with the given name.
     * @param resourceName the path to resource name relative to the classloader
root path.
     *        If <code>resourceName</code> starts with "/", remove it.
     * @param primary the primary ClassLoader to use (or null)
     * @return A URL object for reading the resource, or null if the resource could
not be found or
     *         the invoker doesn't have adequate privileges to get the resource.
     * @throws IOException If I/O errors occur
     * @see ClassLoader#getResource(java.lang.String)
     */
    public URL getResource(String resourceName, ClassLoader primary);

    /**
     * Sets the primary classloader for the current thread.
     * The primary should be kept in a ThreadLocal variable.
     * @param primary the primary classloader
     */
    void setPrimary(ClassLoader primary);

    /**
     * Unsets the primary classloader for the current thread
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     */
    void unsetPrimary();

}
```

Be aware that you can extend *org.datanucleus.JDOClassLoaderResolver* if you just want to change some behaviour of the default loader process. Your class loader resolver should provide a constructor taking an argument of type *ClassLoader* which will be the loader that the PersistenceManager is using at initialisation (your class can opt to not use this, but must provide the constructor)

Implementation

Let's suppose you want to provide your own resolver *MyClassLoaderResolver*

```
package mydomain;

import org.datanucleus.ClassLoaderResolver;

public class MyClassLoaderResolver implements ClassLoaderResolver
{
    /**
     * Constructor for PersistenceManager cases.
     * @param pmLoader Loader from PM initialisation time.
     */
    public MyClassLoaderResolver(ClassLoader pmLoader)
    {
         ...
    }

    .. (implement the interface)
}
```

Plugin Specification

When we have defined our "IdentifierFactory" we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.classloader_resolver">
        <class-loader-resolver name="myloader"
class-name="mydomain.MyClassLoaderResolver"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your new *ClassLoaderResolver* plugin. You do this by having your plugin

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

in the CLASSPATH at runtime, and setting the PMF property **datanucleus.classLoaderResolverName** to *myloader* (the name you specified in the plugin.xml file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.5 Datastore Identity

Plugins : Datastore Identity

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the class used to represent datastore-identity (for JDO). DataNucleus provides a default implementation for use and this generates identifiers (returned by *JDOHelper.getObjectId(obj)*) of the form "3286[OID]mydomain.MyClass". Having this component configurable means that you can override the output of the toString() to be more suitable for any use of these identities. Please be aware that the JDO2 specification (5.4.3) has strict rules for datastore identity classes.

DataNucleus allows identities to be either datastore or application identity. When using datastore identity it needs to have a class to represent an identity. DataNucleus provides its own default datastore identity class. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_datastoreidentity*.

Plugin extension-point	Key	Description	Location
org.datanucleus.store_datastoreidentity	datanucleus	Datastore Identity used by DataNucleus since DataNucleus 1.0 ("1[OID]org.datanucleus.myClass")	datanucleus-core
org.datanucleus.store_datastoreidentity	kodo	Datastore Identity in the style of OpenJPA/Kodo ("org.datanucleus.myClass-1")	datanucleus-core
org.datanucleus.store_datastoreidentity	xcalia	Datastore Identity in the style of Xcalia ("org.datanucleus.myClass:1"). This ignores Xcalias support for class aliases	datanucleus-core

The following sections describe how to create your own datastore identity plugin for DataNucleus.

Interface

Any datastore identity plugin will need to implement *org.datanucleus.identity.OID* So you need to implement the following interface

```
import org.datanucleus.identity;

public interface OID
{
    /**
     * Provides the OID in a form that can be used by the database as a key.
     * @return The key value
     */
    public abstract Object getKeyValue();

    /**
     * Accessor for the PC class name
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * @return the PC Class
     */
    public abstract String getPcClass();

    /**
     * Equality operator.
     * @param obj Object to compare against
     * @return Whether they are equal
     */
    public abstract boolean equals(Object obj);

    /**
     * Accessor for the hashcode
     * @return Hashcode for this object
     */
    public abstract int hashCode();

    /**
     * Returns the string representation of the OID.
     * The string representation should contain enough information to be usable as
input to a String constructor
     * to create the OID.
     * @return the string representation of the OID.
     */
    public abstract String toString();
}
```

Implementation

DataNucleus provides an abstract base class org.datanucleus.identity.OIDImpl as a guideline. The DataNucleus internal implementation is defined as

```
package org.datanucleus.identity;

public class OIDImpl implements java.io.Serializable, OID
{
    /** Localiser for messages. */
    protected static final transient Localiser LOCALISER =
Localiser.getInstance("org.datanucleus.store.Localisation");

    /** Separator to use between fields. */
    private transient static final String oidSeparator = "[OID]";

    // JDO spec 5.4.3 - serializable fields required to be public.

    /** The key value. */
    public final Object oid;

    /** The PersistenceCapable class name */
    public final String pcClass;

    /** pre-created toString to improve performance **/
    public final String toString;

    /** pre-created hasCode to improve performance **/
    public final int hashCode;
```

```
    /**
    * Creates an OID with no value. Required by the JDO spec
    */
    public OIDImpl()
    {
        oid = null;
        pcClass = null;
        toString = null;
        hashCode = -1;
    }

    /**
     * Create a string datastore identity.
     * @param pcClass The PersistenceCapable class that this represents
     * @param object The value
     */
    public OIDImpl(String pcClass, Object object)
    {
        this.pcClass = pcClass;
        this.oid = object;

        StringBuffer s = new StringBuffer();
        s.append(this.oid.toString());
        s.append(oidSeparator);
        s.append(this.pcClass);
        toString = s.toString();
        hashCode = toString.hashCode();
    }

    /**
     * Constructs an OID from its string representation that is consistent with the
output of toString().
     * @param str the string representation of an OID
     * @exception IllegalArgumentException if the given string representation is not
valid.
     * @see #toString
     */
    public OIDImpl(String str)
    throws IllegalArgumentException
    {
        if (str.length() < 2)
        {
            throw new
IllegalArgumentException(LOCALISER.msg("OID.InvalidValue",str));
        }

        int start = 0;
        int end = str.indexOf(oidSeparator, start);
        String oidStr = str.substring(start, end);
        Object oidValue = null;
        try
        {
            // Use Long if possible, else String
            oidValue = new Long(oidStr);
        }
        catch (NumberFormatException nfe)
        {
            oidValue = oidStr;
        }
        oid = oidValue;

        start = end + oidSeparator.length();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        this.pcClass = str.substring(start, str.length());

        toString = str;
        hashCode = toString.hashCode();
    }

    /**
     * Accessor for the key value.
     * @return The key value
     */
    public Object getKeyValue()
    {
        return oid;
    }

    /**
     * Accessor for the PersistenceCapable class name.
     * @return PC class name
     */
    public String getPcClass()
    {
        return pcClass;
    }

    /**
     * Equality operator.
     * @param obj Object to compare against
     * @return Whether they are equal
     */
    public boolean equals(Object obj)
    {
        if (obj == null)
        {
            return false;
        }
        if (obj == this)
        {
            return true;
        }
        if (!(obj.getClass().getName().equals(ClassNameConstants.OIDImpl)))
        {
            return false;
        }
        if (hashCode() != obj.hashCode())
        {
            return false;
        }
        return true;
    }

    /**
     * Accessor for the hashcode
     * @return Hashcode for this object
     */
    public int hashCode()
    {
        return hashCode;
    }

    /**
     * Creates a String representation of the datastore identity, formed from the PC
class name
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * and the key value. This will be something like
     * <pre>3254[OID]mydomain.MyClass</pre>
     * @return The String form of the identity
     */
    public String toString()
    {
        return toString;
    }
}
```

As show you need 3 constructors. One is the default constructor. One takes a String (which is the output of the toString() method). The other takes the PC class name and the key value.

Plugin Specification

So once we have our custom "datastore identity" we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_datastoreidentity">
        <datastoreidentity name="myoid" class-name="mydomain.MyOIDImpl"
unique="true"/>
    </extension>
</plugin>
```

The name "myoid" should be specified when you create the PersistenceManagerFactory using the persistence property name "org.datanucleus.datastoreIdentityType". Thats all. You now have a DataNucleus "datastore identity" plugin.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.6 Identity Translator

Plugins : Identity Translators

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is translation of identities. When you call *pm.getObjectById* you pass in an object. You can provide a plugin that translates this object into a valid JDO identity. Alternatively you could do this in your own code, but the facility is provided. This means that in your application you only use your own form of identities.

You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.identity_translator*.

Plugin extension-point	Key	Description	Location
org.datanucleus.identity_translator	xcalia	Translator that allows for {discriminator}:key as well as the usual input, as supported by Xcalia XIC	datanucleus-core

Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.IdentifierFactory*. So you need to implement the following interface

```
package org.datanucleus.identity;

public interface IdentityTranslator
{
    /**
     * Method to translate the object into the identity.
     * @param om ObjectManager
     * @param obj The object
     * @return The identity
     */
    Object getIdentity(ObjectManager om, Object obj);
}
```

Plugin Specification

When we have defined our "IdentityTranslator" we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.identity_translator">
        <identitytranslator name="mytranslator"
class-name="mydomain.MyIdTranslator"/>
    </extension>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
</plugin>
```

Plugin Usage

The only thing remaining is to use your new *IdentityTranslator* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property **datanucleus.identityTranslatorType** to *mytranslator* (the name you specified in the plugin.xml file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.7 Annotations

Plugins : Annotations

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the reading of annotations. DataNucleus provides a support for JDO2 and JPA1 annotations, but is structured so that you can easily add your own annotations and have them usable within your DataNucleus usage.

DataNucleus supports Java5 annotations. More than this, it actually provides a pluggable framework whereby you can plug in your own annotations support. The Java5 plugin provides plugins for JDO2 and JPA1 annotations. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.annotations*.

Plugin extension-point	Key	Description	Location
org.datanucleus.annotations	@PersistenceCapable	JDO2 annotation reader	datanucleus-core
org.datanucleus.annotations	@PersistenceAware	JDO2 annotation reader	datanucleus-core
org.datanucleus.annotations	@Entity	JPA1 annotation reader	datanucleus-jpa
org.datanucleus.annotations	@MappedSuperclass	JPA1 annotation reader	datanucleus-jpa
org.datanucleus.annotations	@Embeddable	JPA1 annotation reader	datanucleus-jpa

Interface

Any annotation reader plugin will need to implement *org.datanucleus.metadata.annotations.AnnotationReader*. So you need to implement the following interface

```
package org.datanucleus.metadata.annotations;

import org.datanucleus.metadata.PackageMetaData;
import org.datanucleus.metadata.ClassMetaData;

public interface AnnotationReader
{
    /**
     * Accessor for the annotations packages supported by this reader.
     * @return The annotations packages that will be processed.
     */
    String[] getSupportedAnnotationPackages();

    /**
     * Method to get the ClassMetaData for a class from its annotations.
     * @param cls The class
     * @param pmd MetaData for the owning package (that this will be a child of)
     * @return The ClassMetaData (unpopulated and unitialised)
     */
    public ClassMetaData getMetaDataForClass(Class cls, PackageMetaData pmd);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
}
```

Plugin Specification

So we now have our custom "annotation reader" and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.annotations" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.annotations">
        <annotations annotation-class="mydomain.annotations.MyAnnotationType"
                reader="mydomain.annotations.MyAnnotationReader"/>
    </extension>
</plugin>
```

So here we have our "annotations reader" class "MyAnnotationReader" which will process any classes annotated with the annotation "MyAnnotationType".

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.8 MetaData Handler

Plugins : MetaData Handler

DataNucleus has supported JDO metadata from the outset. More than this, it actually provides a pluggable framework whereby you can plug in your own MetaData support. DataNucleus provides JDO and JPA metadata support, as well as "persistence.xml". You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.metadata_handler*.

Plugin extension-point	Key	Description	Location
org.datanucleus.metadata_handler	jdo	JDO MetaData handler	datanucleus-core
org.datanucleus.metadata_handler	persistence	"persistence.xml" MetaData handler	datanucleus-core
org.datanucleus.metadata_handler	jpa	JPA MetaData handler	datanucleus-jpa

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.9 MetaData Entity Resolver

Plugins : MetaData Entity Resolver

DataNucleus provides a pluggable framework whereby you can plug in your own DTD or XSD schema support. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.metadata_entityresolver*.

An Entity Resolver extension point configuration is a list of XSDs schemas or DTDs. For XSD schemas, it needs only the schema location (URL in classpath), since the XML parser will take care of association of XML elements and namespaces to the schemas types. For DTDs, it's a little less obvious since DTDs are not namespace aware, so the configuration must contain the URL location (location in classpath), DTD DOCTYPE (PUBLIC or SYSTEM), and the DTD identity.

An Entity Resolver has close relationship to the MetaData Handler extension point, since the MetaData Handler uses the Entity Resolvers to validate XML files. However, the Entity Resolvers configured via this plugin are only applicable to the *org.datanucleus.metadata.xml.PluginEntityResolver* instances, so the MetaData Handler must use the *PluginEntityResolver* Entity Resolver if it wants to use the schemas configured in this extension point.

Implementation

The Entity Resolver implementation must extend the *org.datanucleus.util.AbstractXMLEntityResolver* class and must have a public constructor that takes the *org.datanucleus.plugin.PluginManager* class.

```
import org.datanucleus.plugin.PluginManager;
import org.datanucleus.util.AbstractXMLEntityResolver;

/**
 * Implementation of an entity resolver for DTD/XSD files.
 * Handles entity resolution for files configured via plugins.
 */
public class PluginEntityResolver extends AbstractXMLEntityResolver
{
    public PluginEntityResolver(PluginManager pm)
    {
                publicIdEntities.put("identity...", "url....");
                ...
                systemIdEntities.put("identity...", "url....");
                ...
        }
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.10 Value Generators

Plugins : Value Generators

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the generation of identity or field values. DataNucleus provides a large selection of generators but is structured so that you can easily add your own variant and have it usable within yourDataNucleus usage.

DataNucleus provides a series of generators for identities/values of fields. The JDO2/JPA1 specs define various that are required. DataNucleus provides plugins for many generators. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_valuegenerator*.

Plugin extension-point	Key	Datastore	Description	Location
org.datanucleus.store_valuegenerator	auid	all datastores	Value Generator using AUIDs	datanucleus-core
org.datanucleus.store_valuegenerator	uuid-hex	all datastores	Value Generator using uuid-hex	datanucleus-core
org.datanucleus.store_valuegenerator	uuid-string	all datastores	Value Generator using uuid-string	datanucleus-core
org.datanucleus.store_valuegenerator	timestamp	all datastores	Value Generator using Timestamp	datanucleus-core
org.datanucleus.store_valuegenerator	timestamp-value	all datastores	Value Generator using Timestamp millisecs value	datanucleus-core
org.datanucleus.store_valuegenerator	increment	rdbms	Value Generator using increment strategy	datanucleus-rdbms
org.datanucleus.store_valuegenerator	sequence	rdbms	Value Generator using datastore sequences	datanucleus-rdbms
org.datanucleus.store_valuegenerator	table-sequence	rdbms	Value Generator using a database table to generate sequences (same as increment)	datanucleus-rdbms
org.datanucleus.store_valuegenerator	max	rdbms	Value Generator using max(COL)+1 strategy	datanucleus-rdbms
org.datanucleus.store_valuegenerator	datastore-uuid-hex	rdbms	Value Generator using uuid-hex attributed by the datastore	datanucleus-rdbms
org.datanucleus.store_valuegenerator	increment	db4o	Value Generator using increment strategy	datanucleus-db4o
org.datanucleus.store_valuegenerator	sequence	db4o	Value Generator using datastore sequences	datanucleus-db4o

The following sections describe how to create your own value generator plugin for DataNucleus.

Interface

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Any value generator plugin will need to implement *org.datanucleus.store.valuegenerator.ValueGenerator* So you need to implement the following interface

```
public interface ValueGenerator
{
    String getName ();

    void allocate (int additional);

    Object next ();
    Object current ();

    long nextValue();
    long currentValue();
}
```

Implementation

DataNucleus provides an abstract base class org.datanucleus.store.valuegenerator.AbstractValueGenerator to extend if you don't require datastore access. If you do require (RDBMS) datastore access for your ValueGenerator then you can extend org.datanucleus.store.rdbms.valuegenerator.AbstractRDBMSValueGenerator Let's give an example, here we want a generator that provides a form of UUID identity. We define our class as

```
package mydomain;

import org.datanucleus.store.valuegenerator.ValueGenerationBlock;
import org.datanucleus.store.valuegenerator.AbstractValueGenerator;

public class MyUUIDValueGenerator extends AbstractValueGenerator
{
    public MyUUIDValueGenerator(String name, Properties props)
    {
        super(name, props);
    }

    /**
     * Method to reserve "size" ValueGenerations to the ValueGenerationBlock.
     * @param size The block size
     * @return The reserved block
     */
    public ValueGenerationBlock reserveBlock(long size)
    {
        Object[] ids = new Object[(int) size];
        for (int i = 0; i < size; i++)
        {
            ids[i] = getIdentifier();
        }
        return new ValueGenerationBlock(ids);
    }

    /**
     * Create a UUID identifier.
     * @return The identifier
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     */
    private String getIdentifier()
    {
        ... Write this method to generate the identifier
    }
}
```

As show you need a constructor taking 2 arguments *String* and *java.util.Properties*. The first being the name of the generator, and the second containing properties for use in the generator.

- **class-name** Name of the class that the value is being added to
- **root-class-name** Name of the root class in this inheritance tree
- **field-name** Name of the field whose value is being set (not provided if this is datastore identity field)
- **catalog-name** Catalog that objects of the class are stored in
- **schema-name** Schema that objects of the class are stored in
- **table-name** Name of the (root) table storing this field
- **column-name** Name of the column storing this field
- **sequence-name** Name of the sequence (if specified in the MetaData)

Plugin Specification

So we now have our custom "value generator" and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_valuegenerator">
        <valuegenerator name="myuuid" class-name="mydomain.MyUUIDValueGenerator"
unique="true"/>
    </extension>
</plugin>
```

The name "myuuid" is what you will use as the "strategy" when specifying to use it in MetaData. The flag "unique" is only needed if your generator is to be unique across all requests. For example if your generator was only unique for a particular class then you should omit that part. Thats all. You now have a DataNucleus "value generator" plugin.

Plugin Usage

To use your value generator you would reference it in your JDO MetaData like this

```
<class name="MyClass">
    <datastore-identity strategy="myuuid"/>
    ...
</class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Don't forget that if you write a value generator that could be of value to others you could easily donate it to DataNucleus for inclusion in the next release.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.11 Level 1 Cache

Plugins : Level 1 Cache

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the Level 1 caching of objects (between PersistenceManagers for the same PersistenceManagerFactory). The Cache guide (JDO or JPA) defines three Level 1 caches but is a plugin point so that you can easily add your own variant and have it usable within your DataNucleus usage.

DataNucleus is able to support third party Level 1 Cache products. There are DataNucleus-provided plugins for weak and soft referenced caches. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.cache_level1*.

Plugin extension-point	Key	Description	Location
org.datanucleus.cache_level1	weak	Weak referenced cache (default)	datanucleus-core
org.datanucleus.cache_level1	soft	Soft referenced cache	datanucleus-core
org.datanucleus.cache_level1	hard	Hard-referenced cache (HashMap)	datanucleus-core

The following sections describe how to create your own Level 1 cache plugin for DataNucleus.

Interface

If you have your own Level1 cache you can easily use it with DataNucleus. DataNucleus defines a Level1Cache interface and you need to implement this.

```
package org.datanucleus.cache;

public interface Level1Cache extends Map
{
}
```

So you need to create a class, **MyLevel1Cache** for example, that implements this interface (i.e that implements *java.util.Map*).

Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file *plugin.xml* in your JAR at the root of the CLASSPATH. The file *plugin.xml* will look like this

```
<?xml version="1.0"?>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<plugin id="mydomain.mycache" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.cache_level1">
            <cache name="MyCache" class-name="mydomain.MyLevel1Cache"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your L1 Cache plugin. To do this you specify the PersistenceManagerFactory property *datanucleus.cache.level1.type* as **MyCache** (the "name" in *plugin.xml*).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.12 Level 2 Cache

Plugins : Level 2 Cache

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the Level 2 caching of objects (between PersistenceManagers for the same PersistenceManagerFactory). The Cache guide (JDO or JPA) defines a large selection of Level 2 caches (builtin, Coherence, EHCache, OSCache, SwarmCache) but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

DataNucleus is able to support third party Level 2 Cache products. There are provided plugins for EHCache, SwarmCache, OSCache, and Oracle Coherence. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.cache_level2*.

Plugin extension-point	Key	Description	Location
org.datanucleus.cache_level2	default	Level 2 Cache (default)	datanucleus-core
org.datanucleus.cache_level2	soft	Level 2 Cache using Soft maps	datanucleus-core
org.datanucleus.cache_level2	ehcache	Level 2 Cache using EHCache	datanucleus-cache
org.datanucleus.cache_level2	ehcacheclassbased	Level 2 Cache using EHCache (based on classes)	datanucleus-cache
org.datanucleus.cache_level2	oscache	Level 2 Cache using OSCache	datanucleus-cache
org.datanucleus.cache_level2	swarmcache	Level 2 Cache using SwarmCache	datanucleus-cache
org.datanucleus.cache_level2	coherence	Level 2 Cache using Oracle Coherence	datanucleus-cache

The following sections describe how to create your own Level 2 cache plugin for DataNucleus.

Interface

If you have your own Level2 cache you can easily use it with DataNucleus. DataNucleus defines a Level2Cache interface and you need to implement this.

```
package org.datanucleus.cache;
public interface Level2Cache
{
    void close();

    void evict (Object oid);
    void evictAll ();
    void evictAll (Object[] oids);
    void evictAll (Collection oids);
    void evictAll (Class pcClass, boolean subclasses);

    void pin (Object oid);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    void pinAll (Collection oids);
    void pinAll (Object[] oids);
    void pinAll (Class pcClass, boolean subclasses);

    void unpin(Object oid);
    void unpinAll(Collection oids);
    void unpinAll(Object[] oids);
    void unpinAll(Class pcClass, boolean subclasses);

    int getNumberOfPinnedObjects();
    int getNumberOfUnpinnedObjects();
    int getSize();
    CachedPC get(Object oid);
    CachedPC put(Object oid, CachedPC pc);
    boolean isEmpty();
    void clear();
    boolean containsOid(Object oid);
}
```

Implementation

Let's suppose your want to implement your own Level 2 cache *MyLevel2Cache*

```
package mydomain;

import org.datanucleus.OMFContext;
import org.datanucleus.cache.Level2Cache;

public class MyLevel2Cache implements Level2Cache
{
    /**
     * Constructor.
     * @param omfCtx OMF Context
     */
    public DefaultLevel2Cache(OMFContext omfCtx)
    {
        ...
    }

    ... (implement the interface)
}
```

Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file *plugin.xml* in your JAR at the root of the CLASSPATH. The file *plugin.xml* will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.cache_level2">
            <cache name="MyCache" class-name="mydomain.MyLevel2Cache"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your L2 Cache plugin. To do this you specify the PersistenceManagerFactory property *datanucleus.cache.level2.type* as **MyCache** (the "name" in *plugin.xml*).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.13 Query Language

Plugins : Query Language

DataNucleus provides support for query languages to allow access to the persisted objects. DataNucleus Core supports JDOQL, SQL and JPQL for use with the "rdbms" StoreManager. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_query_query*.

Plugin extension-point	Key	Datastore	Description	Location
org.datanucleus.store_query_query	JDOQL	rdbms	Query support for JDOQL	datanucleus-rdbms
org.datanucleus.store_query_query	SQL	rdbms	Query support for SQL	datanucleus-rdbms
org.datanucleus.store_query_query	JPQL	rdbms	Query support for JPQL	datanucleus-rdbms
org.datanucleus.store_query_query	JDOQL	db4o	Query support for JDOQL	datanucleus-db4o
org.datanucleus.store_query_query	SQL	db4o	Query support for SQL	datanucleus-db4o
org.datanucleus.store_query_query	Native	db4o	DB4O Native query support	datanucleus-db4o
org.datanucleus.store_query_query	JDOQL	ldap	Query support for JDOQL	datanucleus-ldap
org.datanucleus.store_query_query	JDOQL	excel	Query support for JDOQL	datanucleus-excel
org.datanucleus.store_query_query	JDOQL	xml	Query support for JDOQL	datanucleus-xml
org.datanucleus.store_query_query	Native	neodatis	NeoDatis Native query support	datanucleus-neodatis
org.datanucleus.store_query_query	Criteria	neodatis	NeoDatis Criteria query support	datanucleus-neodatis
org.datanucleus.store_query_query	JDOQL	neodatis	Query support for JDOQL	datanucleus-neodatis
org.datanucleus.store_query_query	JDOQL	json	Query support for JDOQL	datanucleus-json

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.14 Query Methods

JDOQL/JPQL : Query Methods

JDOQL/JPQL are defined to support particular methods/functions as part of the supported syntax. This support is provided by way of an extension point, with support for these methods/functions added via extensions. You can make use of this extension point to add on your own methods/functions - obviously this will be DataNucleus specific. **This plugin extension point is currently only for evaluation of queries in-memory.** It will have no effect where the query is evaluated in the datastore. The plugin extension used here is *org.datanucleus.store_query_methods*.

Plugin extension-point	Name	Type	Description	Location
org.datanucleus.store_query_methods	Math.abs	(static)	Use of Math functions for JDO2	datanucleus-core
org.datanucleus.store_query_methods	Math.sqrt	(static)	Use of Math functions for JDO2	datanucleus-core
org.datanucleus.store_query_methods	JDOHelper.getObjectId	(static)	Use of JDOHelper functions for JDO2	datanucleus-core
org.datanucleus.store_query_methods	JDOHelper.getVersion	(static)	Use of JDOHelper functions for JDO2	datanucleus-core
org.datanucleus.store_query_methods	CURRENT_DATE	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	CURRENT_TIME	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	CURRENT_TIMESTAMP	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	ABS	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	SQRT	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	MOD	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	SIZE	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	UPPER	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	LOWER	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	LENGTH	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	CONCAT	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	SUBSTRING	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	LOCATE	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	TRIM	(static)	JPQL functions	datanucleus-core
org.datanucleus.store_query_methods	startsWith	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	endsWith	String	JDOQL methods	datanucleus-core

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Plugin extension-point	Name	Type	Description	Location
org.datanucleus.store_query_methods	indexOf	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	substring	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	toUpperCase	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	toLowerCase	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	matches	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	trim	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	length	String	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	size	Collection/Map	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	isEmpty	Collection/Map	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	contains	Collection	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	containsKey	Map	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	containsValue	Map	JDOQL methods	datanucleus-core
org.datanucleus.store_query_methods	get	Map	JDOQL methods	datanucleus-core

Interface

Any query method/function plugin will need to implement *org.datanucleus.query.evaluator.memory.InvocationEvaluator*. So you need to implement the following interface

```
public interface InvocationEvaluator
{
    /**
     * Method to evaluate the InvokeExpression, as part of the overall evaluation
     * defined by the InMemoryExpressionEvaluator.
     * @param expr The expression for invocation
     * @param invokedValue Value on which we are invoking this method
     * @param eval The overall evaluator for in-memory
     * @return The result
     */
    Object evaluate(InvokeExpression expr, Object invokedValue,
InMemoryExpressionEvaluator eval);

    /**
     * Accessor for whether this evaluator supports invocation of this method
     * on the specified type.
     * @param cls Type to check
     * @return Whether it is supported.
     */
    boolean supportsType(Class cls);
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Implementation

Let's assume that you want to provide your own method for "String" *toUpperCase* [*obviously this is provided out of the box, but is here as an example*].

```
public class StringToUpperCaseMethodEvaluator implements InvocationEvaluator
{
    protected static final Localiser LOCALISER = Localiser.getInstance(
        "org.datanucleus.Localisation",
ObjectManagerFactoryImpl.class.getClassLoader());

    public Object evaluate(InvokeExpression expr, Object invokedValue,
InMemoryExpressionEvaluator eval)
    {
        String method = expr.getOperation(); // Will be "toUpperCase"

        if (invokedValue == null)
        {
            return null;
        }
        if (!(invokedValue instanceof String))
        {
            throw new NucleusException(LOCALISER.msg("021011",
                method, invokedValue.getClass().getName()));
        }
        return ((String)invokedValue).toUpperCase();
    }

    public boolean supportsType(Class cls)
    {
        if (cls == null)
        {
            // If cls is null we return false since not a STATIC method
            // If your method is STATIC return true
            return false;
        }
        // Applies to String fields
        return String.class.isAssignableFrom(cls);
    }
}
```

Plugin Specification

When we have defined our query method we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root and add a MANIFEST.MF as per the Plugins Guide. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_query_methods">
        <query-method name="toUpperCase"
memory-evaluator="org.datanucleus.query.evaluator.memory.StringToUpperCaseMethodEvaluator"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your method in a JDOQL/JPQL query, like this

```
Query q = pm.newQuery("SELECT FROM mydomain.Product WHERE name.toUpperCase() ==
'KETTLE'");
```

so when evaluating the query in memory it will call this evaluator class for the field 'name'.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.15 Class Enhancers

Plugins : Class Enhancer

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the enhancer of classes.

DataNucleus relies on byte-code enhancement of classes to implement the *PersistenceCapable* and *Detachable* interfaces. It provides a class enhancer plugin using ObjectWeb ASM. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.enhancer.enhancer*.

Plugin extension-point	Key	Description	Location
org.datanucleus.enhancer.enhancer	asm	Class Enhancer using ASM	datanucleus-enhancer

Interface

Any class enhancer plugin will need to implement *org.datanucleus.enhancer.ClassEnhancer*. So you need to implement the following interface

```
package org.datanucleus.enhancer;

public interface ClassEnhancer
{
    /**
     * Check whether the class is enhanced.
     * @return Whether the class is already enhanced.
     */
    boolean checkEnhanced();

    /**
     * Method to enhance the class definition internally.
     * @return Whether the class was enhanced successfully
     */
    boolean enhance();

    /**
     * Method to save the (current) class definition bytecode into a class file.
     * Only has effect if the bytecode has been modified (by enhance()).
     * If directoryName is specified it will be written to
$directoryName/className.class
     * else will overwrite the existing class.
     * @param directoryName Name of a directory (or null to overwrite the class)
     * @throws IOException If an I/O error occurs in the write.
     */
    void save(String directoryName) throws IOException;

    /**
     * Access the class bytecode.
     * @return the class in byte array format
     */
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    byte[] getBytes();

    /**
     * Method to verify the enhancement state.
     * @throws Exception Thrown if an error occurs while verifying
     **/
    void verify() throws Exception;

    /**
     * Accessor for the ClassLoaderResolver in use.
     * @return ClassLoader resolver
     */
    ClassLoaderResolver getClassLoaderResolver();

    /**
     * Accessor for the ClassMetaData for the class.
     * @return MetaData for the class
     */
    ClassMetaData getClassMetaData();
}
```

Be aware that you can extend *org.datanucleus.enhancer.AbstractClassEnhancer*.

Plugin Specification

When we have defined our "ClassEnhancer" we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.enhancer.enhancer">
        <class-enhancer name="MyEnhancer" api="jdo"
class-name="mydomain.MyClassEnhancer"
            test-class="mydomain.SomeClassInTheClasspath"/>
    </extension>
</plugin>
```

The *test-class* should be the name of a class that needs to be in the CLASSPATH when running your enhancer. For example, with ASM we specify *org.objectweb.asm.Opcodes*. The *api* is the API that your enhancer is to be run against. You can specify it for JDO and JPA by adding two lines above (one for each).

Plugin Usage

The only thing remaining is to use your new *ClassEnhancer* plugin. You do this by having your plugin in the CLASSPATH at runtime, and passing the argument **enhancerName** with value *MyEnhancer* (the name you specified in the plugin.xml file) to the DataNucleus Enhancer.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.16 Implementation Creator

Plugins : Implementation Creator

Implementation Creator is an extension point that allows the persistence of interfaces. It is responsible for constructing persistent classes out of interfaces at runtime. The DataNucleus Enhancer plug-in has two ImplementationCreator implementations one using BCEL and the other using ASM. You must have the DataNucleus Enhancer plugin in the CLASSPATH together with either BCEL or ASM whichever you select. The ImplementationCreator is declared via the extension point *org.datanucleus.implementation_creator* and you can use the PMF setting *org.datanucleus.implementationCreatorName* to select which one you want to use. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.implementation_creator*.

Plugin extension-point	Key	Description	Location
org.datanucleus.implementation_creator	asm	Implementation Creator using ASM	datanucleus-enhancer
org.datanucleus.implementation_creator	bcel	Implementation Creator using BCEL	datanucleus-enhancer

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.17 Management Server

Plugins : Management Server

DataNucleus exposes runtime metrics via JMX MBeans. Management Servers permits DataNucleus to register its MBeans into a different MBeanServer. Management Servers can be plugged using the plugin extension *org.datanucleus.management_server*. The following plugin extensions are currently available

Plugin extension-point	Key	Description	Location
org.datanucleus.management_server	default	Register DataNucleus MBeans into the JVM MBeanServer.	datanucleus-management
org.datanucleus.management_server	mx4j	Register DataNucleus MBeans into a MX4J MBeanServer.	datanucleus-mx4j

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

17.18 JTA Locator

Plugins : JTA Locator

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the locator for JTA TransactionManagers (since J2EE doesnt define a standard mechanism for location). DataNucleus provides several plugins for the principal application servers available but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

Locators for JTA TransactionManagers can be plugged using the plugin extension *org.datanucleus.jta_locator*. These are of relevance when running with JTA transaction and linking in to the JTA transaction of some controlling application server

Plugin extension-point	Key	Description	Location
org.datanucleus.jta_locator	jboss	JBoss	datanucleus-core
org.datanucleus.jta_locator	jonas	JOnAS	datanucleus-core
org.datanucleus.jta_locator	jotm	JOTM	datanucleus-core
org.datanucleus.jta_locator	oc4j	OC4J	datanucleus-core
org.datanucleus.jta_locator	orion	Orion	datanucleus-core
org.datanucleus.jta_locator	resin	Resin	datanucleus-core
org.datanucleus.jta_locator	sap	SAP app server	datanucleus-core
org.datanucleus.jta_locator	sun	Sun ONE app server	datanucleus-core
org.datanucleus.jta_locator	weblogic	WebLogic app server	datanucleus-core
org.datanucleus.jta_locator	websphere	WebSphere 4/5	datanucleus-core
org.datanucleus.jta_locator	custom_jndi	Custom JNDI	datanucleus-core

The following sections describe how to create your own JTA Locator plugin for DataNucleus.

Interface

If you have your own JTA Locator you can easily use it with DataNucleus. DataNucleus defines a TransactionManagerLocator interface and you need to implement this.

```
package org.datanucleus.jta;

import javax.transaction.TransactionManager;
import org.datanucleus.ClassLoaderResolver;

public interface TransactionManagerLocator
{
    /**
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * Method to return the TransactionManager.
     * @param clr ClassLoader resolver
     * @return The TransactionManager
     */
    TransactionManager getTransactionManager(ClassLoaderResolver clr);
}
```

So you need to create a class, **MyTransactionManagerLocator** for example, that implements this interface.

Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file *plugin.xml* in your JAR at the root of the CLASSPATH. The file *plugin.xml* will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mylocator" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.jta_locator">
            <cache name="MyLocator"
class-name="mydomain.MyTransactionManagerLocator"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your JTA Locator plugin. To do this you specify the PersistenceManagerFactory property *datanucleus.jtaLocator* as **MyLocator** (the "name" in *plugin.xml*).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.1 JDOQL - User Methods

JDOQL : User Defined Functions for RDBMS

JDOQL is defined by the JDO2 specification and is explicit about what is supported. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_expression_scalarexpression*. This is only for RDBMS

Plugin extension-point	Key	Description	Location
org.datanucleus.store_expression_scalarexpression	java.lang.Math	Use of Math functions for JDO2	datanucleus-rdbms
org.datanucleus.store_expression_scalarexpression	javax.jdo.JDOHelper	Use of JDOHelper function for JDO2	datanucleus-rdbms
org.datanucleus.store_expression_scalarexpression	Analysis	Use of Analysis functions	datanucleus-rdbms

When the JDOQL does not fulfill the querying needs, DataNucleus permits the user to extend it with user-defined methods. The below example shows how to extend the JDOQL language to invoke a custom operation *Finance.pmt(..)*. This custom method calculates payment amounts at interest rate per month.

```
package org.datanucleus.samples.finance;

/**
 * Finance operations
 */
public class Finance
{
    /**
     * Calculate payment amount at interest rate per month
     * @param amount borrowed ammount
     * @param rate rate per payment basis (0.66 month = 8% year)
     * @param payments number of payments
     * @return
     */
    public static double pmt(double amount,double rate,int payments)
    {
        double r = rate / 100;
        return ( amount * rate ) * (Math.pow(1+r, payments) / (Math.pow(1+r,
payments)-1));
    }
}
```

With the above class, we will mirror the code into expressions to be evaluted by JDOQL compiler. We have to implement a ScalarExpression class to generate the SQL statement. The rules are:

- Extends *org.datanucleus.store.expression.ScalarExpression*.
- Add constructor with *org.datanucleus.store.expression.QueryExpression* argument.
- The method to be invoked must return a *org.datanucleus.store.expression.ScalarExpression* instance.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- If the method received arguments, it must be type of *org.datanucleus.store.expression.ScalarExpression*.
- The method name must be postfixed with the word *Method*.

```
package org.datanucleus.samples.finance.expression;

import org.datanucleus.store.expression.NumericExpression;
import org.datanucleus.store.expression.QueryExpression;
import org.datanucleus.store.expression.ScalarExpression;
import org.datanucleus.store.mapping.JavaTypeMapping;

/**
 * Represents expressions of Finance
 */
public class FinanceExpression extends ScalarExpression
{

    /**
     * @param qs The query statement
     */
    protected FinanceExpression(QueryExpression qs)
    {
        super(qs);
    }

    /**
     * Calculate payment amount at interest rate per month
     * Generates a SQL like "(( amount * rate ) * (POWER(1+(rate/100), payments) /
     *                         (POWER(1+(rate/100), payments)-1)))"
     * @param amount borrowed ammount
     * @param rate rate per payment basis (0.66 month = 8% year)
     * @param payments number of payments
     * @return
     */
    public ScalarExpression pmtMethod(ScalarExpression amount, ScalarExpression
rate,
        ScalarExpression payments)
    {
        /*
         * double r = rate / 100;
         * double numerator = (Math.pow(1+r, payments)
         * double denominator = (Math.pow(1+r, payments)-1
         * return ( amount * rate ) * (numerator / denominator)
         */

        //define literal 100
        JavaTypeMapping m100 =
qs.getStoreManager().getDatastoreAdapter().getMapping(Integer.class,
            qs.getStoreManager(),qs.getClassLoaderResolver());
        ScalarExpression literal100 = m100.newLiteral(qs, new Integer(100));

        //define literal 1
        JavaTypeMapping m1 =
qs.getStoreManager().getDatastoreAdapter().getMapping(Integer.class,
            qs.getStoreManager(),qs.getClassLoaderResolver());
        ScalarExpression literal1 = m1.newLiteral(qs, new Integer(1));

        //double r = rate / 100;
        ScalarExpression r = rate.div(literal100).encloseWithInParentheses();

        //double numerator = (Math.pow(1+r, payments)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        ScalarExpression numerator = power(literal1.add(r),
payments).encloseWithInParentheses();

        //double denominator =  (Math.pow(1+r, payments)-1
        ScalarExpression denominator =
           power(literal1.add(r),
payments).sub(literal1).encloseWithInParentheses();

        // return ( amount * rate ) * (numerator / denominator)
        return amount.mul(rate).mul(
numerator.div(denominator).encloseWithInParentheses()).encloseWithInParentheses();
    }

    /**
     * Creates the expression POWER(expr1,expr2). This method assume the database
supports the
     * function POWER.
     * @param expr1 the first argument
     * @param expr2 the second argument
     * @return the result
     */
    private ScalarExpression power(ScalarExpression expr1,ScalarExpression expr2)
    {
        ArrayList args = new ArrayList();
        args.add(expr1);
        args.add(expr2);

        return new NumericExpression("POWER", args);
    }
}
```

The FinanceExpression is registered by declaring it in the /plugin.xml file. The below code exemplifies it:

```
<?xml version="1.0"?>
<plugin id="org.datanucleus" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_expression_scalarexpression">
        <scalar-expression literal-class="org.datanucleus.samples.finance.Finance"
scalar-expression-class="org.datanucleus.samples.finance.expression.FinanceExpression"/>
    </extension>
</plugin>
```

After that pack all the classes and the /plugin.xml into a jar and place it in the classpath.

In JDOQL invoke it with *Literal.method(args)*. The below exemplifies:

```
Query query = pm.newQuery(org.datanucleus.samples.store.Payment.class);
query.setFilter("amount>Finance.pmt(100000,0.66,360)");
List results = (List)query.execute();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.2 Java Type Mapping

Plugins : RDBMS Java Types

When persisting a class to an RDBMS datastore there is a *mapping* process from class/field to table/column. DataNucleus provides a mapping process and allows users to define their own mappings where required. Each field is of a particular type, and where a field is to be persisted as a second-class object you need to define a mapping. DataNucleus defines mappings for all of the required JDO/JPA types but you want to persist some of your own types as second-class objects. This extension is not required for other datastores, just RDBMS.

A type mapping defines the way to convert between an object of the Java type and its datastore representation (namely a column or columns in a datastore table). There are 2 types of Java types that can be mapped. These are mutable (something that can be updated, such as java.util.Date) and immutable (something that is fixed from the point of construction, such as java.awt.Color).

The examples in this guide relate to current DataNucleus SVN. Please consult the DataNucleus source code if you are using an earlier version since things have changed recently for user type mapping

Java type to single column mapping

To map any Java type to a datastore column you need this mapping class we've mentioned. The simplest way to describe how to define your own mapping is to give an example. Lets assume we have our own class **IPAddress** that represents an address such as 192.168.1.1 and we want to map this to a single VARCHAR column in the datastore

```
public class IPAddress
{
    String ipAddress;

    public IPAddress(String ipAddr)
    {
        ipAddress = ipAddr;
    }

    public String toString()
    {
        return ipAddress;
    }
}
```

We start by defining a *JavaTypeMapping* for this type.

```
import org.datanucleus.store.mapped.mapping.ObjectAsStringMapping;

public class IPAddressMapping extends ObjectAsStringMapping
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    private static IPAddress mappingSampleValue = new IPAddress("192.168.1.1");

    public Object getSampleValue(ClassLoaderResolver clr)
    {
        return mappingSampleValue;
    }

    /**
     * Method to return the Java type being represented
     * @return The Java type we represent
     */
    public Class getJavaType()
    {
        return IPAddress.class;
    }

    /**
     * Method to return the default length of this type in the datastore.
     * An IP address can be maximum of 15 characters ("WWW.XXX.YYY.ZZZ")
     * @return The default length
     */
    public int getDefaultLength(int index)
    {
        return 15;
    }

    /**
     * Method to set the datastore string value based on the object value.
     * @param object The object
     * @return The string value to pass to the datastore
     */
    protected String objectToString(Object object)
    {
        String ipaddr;
        if (object instanceof IPAddress)
        {
            ipaddr = ((IPAddress)object).toString();
        }
        else
        {
            ipaddr = (String)object;
        }
        return ipaddr;
    }

    /**
     * Method to extract the objects value from the datastore string value.
     * @param datastoreValue Value obtained from the datastore
     * @return The value of this object (derived from the datastore string value)
     */
    protected Object stringToObject(String datastoreValue)
    {
        return new IPAddress(datastoreValue.trim());
    }
}
```

We have extended the DataNucleus convenience class *ObjectAsStringMapping* and this provides the majority of the mapping for us (including JDOQL capabilities). We have defined a default length of 15 so

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

that when an IP Address is to be mapped, if the user doesnt specify a length in the metadata it will be assigned a column of length 15. Finally we define the methods of converting the IPAddress into a String and back again.

The only remaining thing we need to do is enable use of this Java type when running DataNucleus. To do this we create a "plugin.xml" at the root of the CLASSPATH, like this.

```
<?xml version="1.0"?>
<plugin>
    <extension point="org.datanucleus.store_mapping">
        <mapping java-type="mydomain.IPAddress"
mapping-class="mydomain.IPAddressMapping"/>
    </extension>
</plugin>
```

When using the DataNucleus Enhancer, SchemaTool or Core, DataNucleus automatically searches for the *mapping definition* at **/plugin.xml** files in the classpath.

The most common user-type will map to a String and so can just take this as a template. If the user-type maps to a Long then they can use *ObjectAsLongMapping* - the other convenience mapping available. Note that you also need a MANIFEST.MF as per the Plugins Guide.

This is downloadable as a DataNucleus sample from the Download Page

From this point onwards once you have a field of that type you simply specify the field as this

```
<field name="myfield" persistence-modifier="persistent"/>
```

and it will be persisted using your user-type mapping.

Java Type to multiple column mapping

To map any Java type to multiple datastore columns you need also mapping class. The simplest way to describe how to define your own mapping is to give an example. Here we'll use the examle of the Java AWT class Color. This has 3 colour components (red, green, and blue) as well as an alpha component. So here we want to map the Java type java.awt.Color to 4 datastore columns - one for each of the red, green, blue, and alpha components of the colour. To do this we define a mapping class extending the DataNucleus class *org.datanucleus.store.mapped.mapping.SingleFieldMultiMapping*

```
package org.mydomain;

import org.datanucleus.PersistenceManager;
import org.datanucleus.metadata.AbstractPropertyMetaData;
import org.datanucleus.store.mapped.DatastoreContainerObject;
import org.datanucleus.store.mapped.mapping.SingleFieldMultiMapping;
import org.datanucleus.store.mapped.mapping.JavaTypeMapping;
import org.datanucleus.store.mapped.query.QueryStatement;
import org.datanucleus.store.mapped.expression.TableExpression;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class ColorMapping extends SingleFieldMultiMapping
{
    /**
     * Initialize this JavaTypeMapping with the given DatastoreAdapter for
     * the given FieldMetaData.
     *
     * @param dba The Datastore Adapter that this Mapping should use.
     * @param fmd FieldMetaData for the field to be mapped (if any)
     * @param container The datastore container storing this mapping (if any)
     * @param clr the ClassLoaderResolver
     */
    public void initialize(DatastoreAdapter dba, AbstractMemberMetaData fmd,
                    DatastoreContainerObject container, ClassLoaderResolver clr)
    {
                super.initialize(dba, fmd, container, clr);

        addDatastoreField(ClassNameConstants.INT); // Red
        addDatastoreField(ClassNameConstants.INT); // Green
        addDatastoreField(ClassNameConstants.INT); // Blue
        addDatastoreField(ClassNameConstants.INT); // Alpha
    }

    public Class getJavaType()
    {
        return Color.class;
    }

    public Object getSampleValue(ClassLoaderResolver clr)
    {
        return java.awt.Color.red;
    }

    public void setObject(PersistenceManager pm, Object preparedStatement, int[]
exprIndex, Object value)
    {
        Color color = (Color) value;
        if (color == null)
        {
            getDataStoreMapping(0).setObject(preparedStatement, exprIndex[0], null);
            getDataStoreMapping(1).setObject(preparedStatement, exprIndex[1], null);
            getDataStoreMapping(2).setObject(preparedStatement, exprIndex[2], null);
            getDataStoreMapping(3).setObject(preparedStatement, exprIndex[3], null);
        }
        else
        {
getDataStoreMapping(0).setInt(preparedStatement,exprIndex[0],color.getRed());
getDataStoreMapping(1).setInt(preparedStatement,exprIndex[1],color.getGreen());
getDataStoreMapping(2).setInt(preparedStatement,exprIndex[2],color.getBlue());
getDataStoreMapping(3).setInt(preparedStatement,exprIndex[3],color.getAlpha());
        }
    }

    public Object getObject(PersistenceManager pm, Object resultSet, int[]
exprIndex)
    {
        try
        {
            // Check for null entries
            if (((ResultSet)resultSet).getObject(exprIndex[0]) == null)
            {
                return null;
            }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }
        catch (Exception e)
        {
            // Do nothing
        }

        int red = getDataStoreMapping(0).getInt(resultSet,exprIndex[0]);
        int green = getDataStoreMapping(1).getInt(resultSet,exprIndex[1]);
        int blue = getDataStoreMapping(2).getInt(resultSet,exprIndex[2]);
        int alpha = getDataStoreMapping(3).getInt(resultSet,exprIndex[3]);
        return new Color(red,green,blue,alpha);
    }

    // --------------------------------- JDOQL Query Methods
-------------------------------------------

    public ScalarExpression newLiteral(QueryExpression qs, Object value)
    {
        return null; // Dont support JDOQL querying of Color fields currently
    }

    public ScalarExpression newScalarExpression(QueryExpression qs,
LogicSetExpression te)
    {
        return null; // Dont support JDOQL querying of Color fields currently
    }
}
```

In the initialize() method we've created 4 columns - one for each of the red, green, blue, alpha components of the colour. The argument passed in when constructing these columns is the Java type name of the column data being stored. The other 2 methods of relevance are the setObject() and getObject(). These have the task of mapping between the Color object and its datastore representation (the 4 columns). That's all there is to it.

The above example does not allow the Java type to be used in JDOQL queries. This would involve writing the methods newLiteral(), newScalarExpression() and this detail will be added in a future version of this guide.

The only thing we need to do is enable use of this Java type when running DataNucleus. To do this we create a *plugin.xml* (at the root of our CLASSPATH) to contain our mappings.

```
<?xml version="1.0"?>
<plugin>
    <extension point="org.datanucleus.store_mapping">
        <mapping java-type="java.awt.Color"
mapping-class="org.mydomain.MyColorMapping"/>
    </extension>
</plugin>
```

Note that we also require a MANIFEST.MF file as per the Plugins Guide. When using the DataNucleus Enhancer, SchemaTool or Core, DataNucleus automatically searches for the *mapping definition* at **/plugin.xml** files in the CLASSPATH.

Obviously, since DataNucleus already supports java.awt.Color there is no need to add this particular mapping to DataNucleus yourself, but this demonstrates the way you should do it for any type you wish to add.

If your Java type that you want to map maps direct to a single column then you would instead extend org.datanucleus.store.mapping.SingleFieldMapping and wouldn't need to add the columns yourself. Look at DataNucleus SVN for many examples of doing it this way.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.3 Datastore Mapping

Plugins : RDBMS Datastore Types

When persisting a class to an RDBMS datastore there is a *mapping* process from class/field to table/column. The most common thing to configure is the Java Type Mapping so that it maps between the java type and the column(s) in the desired way. We can also configure the mapping to the datastore type. So, for example, we define what JDBC types we can map a particular Java type to. By "datastore type" we mean the JDBC type, such as BLOB, INT, VARCHAR etc. DataNucleus provides datastore mappings for the vast majority of JDBC types and the handlings will almost always be adequate. What you could do though is make a particular Java type persistable using a different JDBC type if you wished.

Interface

To define your own datastore mapping you need to implement *org.datanucleus.store.mapped.mapping.DatastoreMapping*

```
public interface DatastoreMapping
{
    /**
     * Whether the field mapped is nullable.
     * @return true if is nullable
     */
    boolean isNullable();

    /**
     * The datastore field mapped.
     * @return the DatastoreField
     */
    DatastoreField getDatastoreField();

    /**
     * The mapping for the java type that this datastore mapping is used by.
     * This will return null if this simply maps a datastore field in the datastore
and has
     * no associated java type in a class.
     * @return the JavaTypeMapping
     */
    JavaTypeMapping getJavaTypeMapping();

    /**
     * Accessor for whether the mapping is decimal-based.
     * @return Whether the mapping is decimal based
     */
    boolean isDecimalBased();

    /**
     * Accessor for whether the mapping is integer-based.
     * @return Whether the mapping is integer based
     */
    boolean isIntegerBased();

    /**
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * Accessor for whether the mapping is string-based.
     * @return Whether the mapping is string based
     */
    boolean isStringBased();

    /**
     * Accessor for whether the mapping is bit-based.
     * @return Whether the mapping is bit based
     */
    boolean isBitBased();

    /**
     * Accessor for whether the mapping is boolean-based.
     * @return Whether the mapping is boolean based
     */
    boolean isBooleanBased();

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setBoolean(Object preparedStatement, int paramIndex, boolean value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setChar(Object preparedStatement, int paramIndex, char value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setByte(Object preparedStatement, int paramIndex, byte value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setShort(Object preparedStatement, int paramIndex, short value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setInt(Object preparedStatement, int paramIndex, int value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setLong(Object preparedStatement, int paramIndex, long value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setFloat(Object preparedStatement, int paramIndex, float value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setDouble(Object preparedStatement, int paramIndex, double value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setString(Object preparedStatement, int paramIndex, String value);

    /**
     * Sets a value into preparedStatement
     * at position specified by paramIndex.
     * @param preparedStatement a datastore object that executes statements in the
database
     * @param paramIndex the position of the value in the statement
     * @param value the value
     */
    void setObject(Object preparedStatement, int paramIndex, Object value);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
```

```
    boolean getBoolean(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    char getChar(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    byte getByte(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    short getShort(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    int getInt(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    long getLong(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    float getFloat(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     */
    double getDouble(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    String getString(Object resultSet, int exprIndex);

    /**
     * Obtains a value from resultSet
     * at position specified by exprIndex.
     * @param resultSet an object returned from the datastore with values
     * @param exprIndex the position of the value in the result
     * @return the value
     */
    Object getObject(Object resultSet, int exprIndex);
}
```

So you can define how to convert the datastore column value to/from common Java types. Please refer to the existing types in DataNucleus SVN for examples.

Plugin Specification

To give an example of what the plugin specification looks like

```
<?xml version="1.0"?>
<plugin id="mydomain.myplugins" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.store_datastoremapping">
        <mapping java-type="java.lang.Character"
rdbms-mapping-class="org.datanucleus.store.rdbms.mapping.CharRDBMSMapping"
            jdbc-type="CHAR" sql-type="CHAR" default="true"/>
        <mapping java-type="java.lang.Character"
rdbms-mapping-class="org.datanucleus.store.rdbms.mapping.IntegerRDBMSMapping"
            jdbc-type="INTEGER" sql-type="INT" default="false"/>
    </extension>
</plugin>
```

So in this definition we have defined that a field of type "Character" can be mapped to JDBC type of CHAR or INTEGER (with CHAR the default)

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.4 Datastore Adapter

Plugins : RDBMS Adapters

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the adapter for the datastore. The datastore adapter provides the translation between DataNucleus and the specifics of the RDBMS in use. DataNucleus provides support for a large selection of RDBMS but is structured so that you can easily add your own adapter for your RDBMS and have it usable within your DataNucleus usage.

DataNucleus supports many RDBMS databases, and by default, ALL RDBMS are supported without the need to extend DataNucleus. Due to incompabilities, or specifics of each RDBMS database, it's allowed to extend DataNucleus to make the support to a specific database fit better to DataNucleus and your needs. The RDBMS page lists all RDBMS databases that have been tested with DataNucleus, and some of these databases has been adapted internally to get a good fit. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_datastoreadapter*.

Plugin extension-point	Key	Description	Location
org.datanucleus.store_datastoreadapter	derby	Adapter for Apache Derby/Cloudscape	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	db2	Adapter for IBM DB2	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	as/400	Adapter for IBM DB2 AS/400	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	firebird	Adapter for Firebird/Interbase	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	microsoft	Adapter for MSSQL server	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	h2	Adapter for H2	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	hsql	Adapter for HSQLDB	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	mckoi	Adapter for McKoi DB	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	mysql	Adapter for MySQL	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	sybase	Adapter for Sybase	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	oracle	Adapter for Oracle	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	pointbase	Adapter for Pointbase	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	postgresql	Adapter for PostgreSQL	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	sapdb	Adapter for SAPDB/MaxDB	datanucleus-rdbms
org.datanucleus.store_datastoreadapter	informix	Adapter for Informix	datanucleus-rdbms

DataNucleus supports a very wide range of RDBMS datastores. It will typically auto-detect the datastore adapter to use and select it. This, in general, will work well and the user will not need to change anything to benefit from this behaviour. There are occasions however where a user may need to provide their own datastore adapter and use that. For example if their RDBMS is a new version and something has changed

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

relative to the previous (supported) version, or where the auto-detection fails to identify the adapter since their RDBMS is not yet on the supported list.

By default when you create a PersistenceManagerFactory (PMF) to connect to a particular datastore DataNucleus will automatically detect the datastore adapter to use and will use its own internal adapter for that type of datastore. The default behavious is overridden using the PMF property org.datanucleus.rdbms.datastoreAdapterClassName, which specifies the class name of the datastore adapter class to use. This class must extend the DataNucleus class DatabaseAdapter .

So you need to override the DatabaseAdapter class. You have 2 ways to go here. You can either start from scratch (when writing a brand new adapter), or you can take the existing DataNucleus adapter for a particular RDBMS and change (or extend) it. Let's take an example so you can see what is typically included in such an Adapter. Bear in mind that ALL RDBMS are different in some (maybe small) way, so you may have to specify very little in this adapter, or you may have a lot to specify depending on the RDBMS, and how capable it's JDBC drivers are.

```
public class MySQLAdapter extends DatabaseAdapter
{
    /**
     * A string containing the list of MySQL keywords that are not also SQL/92
     * <i>reserved words</i>, separated by commas.
     */
    public static final String NONSQL92_RESERVED_WORDS =
        "ANALYZE,AUTO_INCREMENT,BDB,BERKELEYDB,BIGINT,BINARY,BLOB,BTREE," +
        "CHANGE,COLUMNS,DATABASE,DATABASES,DAY_HOUR,DAY_MINUTE,DAY_SECOND," +
        "DELAYED,DISTINCTROW,DIV,ENCLOSED,ERRORS,ESCAPED,EXPLAIN,FIELDS," +
        "FORCE,FULLTEXT,FUNCTION,GEOMETRY,HASH,HELP,HIGH_PRIORITY," +
        "HOUR_MINUTE,HOUR_SECOND,IF,IGNORE,INDEX,INFILE,INNODB,KEYS,KILL," +
        "LIMIT,LINES,LOAD,LOCALTIME,LOCALTIMESTAMP,LOCK,LONG,LONGBLOB," +
        "LONGTEXT,LOW_PRIORITY,MASTER_SERVER_ID,MEDIUMBLOB,MEDIUMINT," +
        "MEDIUMTEXT,MIDDLEINT,MINUTE_SECOND,MOD,MRG_MYISAM,OPTIMIZE," +
        "OPTIONALLY,OUTFILE,PURGE,REGEXP,RENAME,REPLACE,REQUIRE,RETURNS," +
        "RLIKE,RTREE,SHOW,SONAME,SPATIAL,SQL_BIG_RESULT,SQL_CALC_FOUND_ROWS," +
        "SQL_SMALL_RESULT,SSL,STARTING,STRAIGHT_JOIN,STRIPED,TABLES," +
        "TERMINATED,TINYBLOB,TINYINT,TINYTEXT,TYPES,UNLOCK,UNSIGNED,USE," +
        "USER_RESOURCES,VARBINARY,VARCHARACTER,WARNINGS,XOR,YEAR_MONTH," +
        "ZEROFILL";

    /**
     * Constructor.
     * Overridden so we can add on our own list of NON SQL92 reserved words
     * which is returned incorrectly with the JDBC driver.
     * @param metadata MetaData for the DB
     **/
    public MySQLAdapter(DatabaseMetaData metadata)
    {
        super(metadata);

        reservedKeywords.addAll(parseKeywordList(NONSQL92_RESERVED_WORDS));
    }

    /**
     * An alias for this adapter.
     * @return The alias
     */
    public String getVendorID()
    {
```

ALL RIGHTS RESERVED

```
        return "mysql";
    }

    /**
     * MySQL, when using AUTO_INCREMENT, requires the primary key specified
     * in the CREATE TABLE, so we do nothing here.
     *
     * @param pkName The name of the primary key to add.
     * @param pk An object describing the primary key.
     * @return The statement to add the primary key separately
     */
    public String getAddPrimaryKeyStatement(SQLIdentifier pkName, PrimaryKey pk)
    {
        return null;
    }

    /**
     * Whether the datastore supports specification of the primary key in
     * CREATE TABLE statements.
     * @return Whetehr it allows "PRIMARY KEY ..."
     */
    public boolean supportsPrimaryKeyInCreateStatements()
    {
        return true;
    }

    /**
     * Method to return the CREATE TABLE statement.
     * Versions before 5 need INNODB table type selecting for them.
     * @param table The table
     * @param columns The columns in the table
     * @return The creation statement
     **/
    public String getCreateTableStatement(TableImpl table, Column[] columns)
    {
        StringBuffer createStmt = new
StringBuffer(super.getCreateTableStatement(table,columns));

        // Versions before 5.0 need InnoDB table type
        if (datastoreMajorVersion < 5)
        {
            createStmt.append(" TYPE=INNODB");
        }

        return createStmt.toString();
    }

    ...
}
```

So here we've shown a snippet from the MySQL DatabaseAdapter. We basically take much behaviour from the base class but override what we need to change for our RDBMS. You should get the idea by now. Just go through the Javadocs of the superclass and see what you need to override.

A final step that is optional here is to integrate your new adapter as a DataNucleus plugin. To do this you need to package it with a file *plugin.xml*, specified at the root of the CLASSPATH. The file should look like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<?xml version="1.0"?>
<plugin id="mydomain" name="MyCompany DataNucleus plug-in"
provider-name="MyCompany">
    <extension point="org.datanucleus.store_datastoreadapter">
        <datastore-adapter vendor-id="myname"
class-name="mydomain.MyDatastoreAdapter" priority="10"/>
    </extension>
</plugin>
```

Where the **myname** specified is a string that is part of the JDBC "product name" (returned by "DatabaseMetaData.getDatabaseProductName()"). If there are multiple adapters for the same *vendor-id* defined, the attribute **priority** is used to determine which one is used. The adapter with the highest number is chosen. Note that the behaviour is undefined when two or more adapters with *vendor-id* have the same priority. All adapters defined in DataNucleus and its official plugins use priority values between **0** and **9**. So, to make sure your adapter is chosen, use a value higher than that.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.5 ConnectionPool

Plugins : Connection Pooling

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the pooling of connections to the datastore. DataNucleus provides a large selection of connection pools (DBCP, C3P0, Proxool) but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

DataNucleus requires a DataSource to define the datastore in use and consequently allows use of connection pooling. DataNucleus provides 3 plugins for different pooling products - DBCP, C3P0, and Proxool. You can easily define your own plugin for pooling. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.datasource*.

Plugin extension-point	Key	Description	Location
org.datanucleus.datasource	c3p0	RDBMS connection pool, using C3P0	datanucleus-connectionpool
org.datanucleus.datasource	dbcp	RDBMS connection pool, using Apache DBCP	datanucleus-connectionpool
org.datanucleus.datasource	proxool	RDBMS connection pool, using Proxool	datanucleus-connectionpool

The following sections describe how to create your own connection pooling plugin for DataNucleus.

Interface

If you have your own DataSource connection pooling implementation you can easily use it with DataNucleus. DataNucleus defines a DataSourceFactory interface and you need to implement this.

```
package org.datanucleus.store.rdbms.datasource;

public interface DataNucleusDataSourceFactory
{
    /**
     * Method to make a DataSource for use within DataNucleus.
     * @param omfCtx OMF Context
     * @return The DataSource
     * @throws Exception Thrown if an error occurs during creation
     */
    public DataSource makePooledDataSource(OMFContext omfCtx);
}
```

Implementation

So let's suppose you have a library (*org.mydomain.MyPoolingClass*) that creates a DataSource that handles

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

pooling. So you would do something like this

```
package mydomain;

import org.datanucleus.ClassLoaderResolver;
import org.datanucleus.store.rdbms.datasource.DataNucleusDataSourceFactory;

public class MyPoolingClassDataSourceFactory implements DataNucleusDataSourceFactory
{
    /**
     * Method to make a DataSource for use within DataNucleus.
     * @param omfCtx OMF Context
     * @return The DataSource
     * @throws Exception Thrown if an error occurs during creation
     */
    public DataSource makePooledDataSource(OMFContext omfCtx)
    {
        PersistenceConfiguration conf = omfCtx.getPersistenceConfiguration();
        String dbDriver =
conf.getStringProperty("datanucleus.ConnectionDriverName");
        String dbURL = conf.getStringProperty("datanucleus.ConnectionURL");
        String dbUser = conf.getStringProperty("datanucleus.ConnectionUserName");
        String dbPassword =
conf.getStringProperty("datanucleus.ConnectionPassword");
        ClassLoaderResolver clr = omfCtx.getClassLoaderResolver(null);

        // Load the database driver
        try
        {
            Class.forName(dbDriver);
        }
        catch (ClassNotFoundException cnfe)
        {
            try
            {
                clr.classForName(dbDriver);
            }
            catch (RuntimeException e)
            {
                // JDBC driver not found
                throw new DatastoreDriverNotFoundException(dbDriver);
            }
        }

        // Check the presence of "mydomain.MyPoolingClass"
        try
        {
            Class.forName("mydomain.MyPoolingClass");
        }
        catch (ClassNotFoundException cnfe)
        {
            try
            {
                clr.classForName("mydomain.MyPoolingClass");
            }
            catch (RuntimeException e)
            {
                // "MyPoolingClass" library not found
                throw new DatastoreLibraryNotFoundException("MyPoolingClass",
"MyPoolingClass");
            }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }

        // Create the Data Source for your pooling library
        // Use the input driver, URL, user/password
        // Use the input configuration file as required
        DataSource ds = new mydomain.MyPoolingClass(...);

        return ds;
    }
}
```

Plugin Specification

The only thing required now is to register this plugin with DataNucleus when you start up your application. To do this create a file *plugin.xml* and put it in your JAR at the root of the CLASSPATH. It should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.datasource">
        <datasource-factory name="MyPoolingClass"
class-name="mydomain.MyPoolingClassDataSourceFactory"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your new *DataNucleusDataSourceFactory* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the persistence property **datanucleus.connectionPoolingType** to *MyPoolingClass* (the name you specified in the plugin.xml file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.6 ConnectionProvider

Plugins : Connection Provider

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the failover mechanism. DataNucleus provides a support for basic Failover algorithm, and is structured so that you can easily add your own failover algorithm and have them usable within your DataNucleus usage.

Failover algorithm for DataNucleus can be plugged using the plugin extension *org.datanucleus.store.connection_provider.*

Plugin extension-point	Key	Description	Location
org.datanucleus.store_connectionprovider	PriorityList	Ordered List Algorithm	datanucleus-rdbms

Interface

Any Connection Provider plugin will need to implement *org.datanucleus.store.rdbms.ConnectionProvider.* So you need to implement the following interface

```
package org.datanucleus.store.rdbms;

import java.sql.Connection;
import java.sql.SQLException;

import javax.sql.DataSource;

/**
 * Connects to a DataSource to obtain a Connection.
 * The ConnectionProvider is not a caching and neither connection pooling mechanism.
 * The ConnectionProvider exists to perform failover algorithm on multiple
DataSources
 * when necessary.
 * One instance per StoreManager (RDBMSManager) is created.
 * Users can provide their own implementation via the extension
org.datanucleus.store_connectionprovider
 */
public interface ConnectionProvider
{
    /**
     * Flag if an error causes the operation to thrown an exception, or false to
skip to next DataSource.
     * If an error occurs on the last DataSource on the list an Exception will be
thrown no matter if
     * failOnError is true or false. This is a hint.
     * Implementations may ignore the user setting and force it's own behaviour
     * @param flag true if to fail on error
     */
    void setFailOnError(boolean flag);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    /**
     * Obtain a connection from the datasources, starting on the first
     * datasource, and if unable to obtain a connection skips to the next one on the
list, and try again
     * until the list is exhausted.
     * @param ds the array of datasources. An ordered list of datasources
     * @return the Connection, null if ds is null, or null if the DataSources has
returned
     *              a null as connection
     * @throws SQLException in case of error and failOnError is true or the error
occurs while obtaining
     *              a connection with the last
     * DataSource on the list
     */
    Connection getConnection(DataSource[] ds) throws SQLException;

}
```

Plugin Specification

So we now have our custom "Connection Provider" and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.connectionprovider" name="My DataNucleus plug-in"
provider-name="MyCompany">
    <extension point="org.datanucleus.store_connectionprovider">
       <connection-provider class-name="mydomain.MyConnectionProvider"
name="MyName"/>
    </extension>
</plugin>
```

So here we have our "ConnectionProvider" class "MyConnectionProvider" which is named "MyName". When constructing the PersistenceManagerFactory, add the setting *datanucleus.rdbms.connectionProviderName=MyName*.

Lifecycle

The *ConnectionProvider* instance is created when the RBMSManager is instantiated and hold as hard reference during the lifecycle of the RDBMSManager.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.7 Identifier Factories

Plugins : Identifier Factories

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the naming of datastore identifiers. DataNucleus provides its own internal identifier factories, but also allows you to plugin your own factory. Identifiers are required when using an RDBMS datastore to define the name of components in the datastore, such as table/columns.

DataNucleus provides the mechanism to generate datastore identifiers (table/column names) when none are defined by the users metadata/annotations. In addition to the default JDO factory there is also an identifier factory that generates identifiers consistent with the JPA1 specification. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.store_identifierfactory*.

Plugin extension-point	Key	Description	Location
org.datanucleus.store_identifierfactory	jpox	Identifier Factory providing DataNucleus JPOX 1.1 default namings	datanucleus-rdbms
org.datanucleus.store_identifierfactory	jpox2	Identifier Factory providing JPOX 1.2 namings	datanucleus-rdbms
org.datanucleus.store_identifierfactory	jpa	Identifier Factory providing JPA-compliant namings	datanucleus-rdbms

Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.IdentifierFactory*. So you need to implement the following interface

```
package org.datanucleus.store;

public interface IdentifierFactory
{
    /** Representation of an identifier specified in UPPER CASE */
    public static final int IDENTIFIER_UPPER_CASE = 0;

    /** Representation of an identifier specified in "UPPER CASE" */
    public static final int IDENTIFIER_UPPER_CASE_QUOTED = 1;

    /** Representation of an identifier specified in lower case. */
    public static final int IDENTIFIER_LOWER_CASE = 2;

    /** Representation of an identifier specified in "lower case" */
    public static final int IDENTIFIER_LOWER_CASE_QUOTED = 3;

    /** Representation of an identifier specified in Mixed Case. */
    public static final int IDENTIFIER_MIXED_CASE = 4;

    /** Representation of an identifier specified in "Mixed Case". */
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    public static final int IDENTIFIER_MIXED_CASE_QUOTED = 5;

    /** identifier for table names **/
    public final static int TABLE = 0;

    /** column **/
    public final static int COLUMN = 1;

    /** foreign key **/
    public final static int FOREIGN_KEY = 2;

    /** index **/
    public final static int INDEX = 3;

    /** candidate key - unique index constraint **/
    public final static int CANDIDATE_KEY = 4;

    /** primary key **/
    public final static int PRIMARY_KEY = 5;

    /** identifier for datastore sequence. */
    public final static int SEQUENCE = 6;

    /**
     * Accessor for the datastore adapter that we are creating identifiers for.
     * @return The datastore adapter
     */
    DatastoreAdapter getDatastoreAdapter();

    /**
     * Accessor for the identifier case being used.
     * @return The identifier case
     */
    int getIdentifierCase();

    /**
     * Convenience method to return the name for the identifier case.
     * @return Identifier case name
     */
    String getNameOfIdentifierCase();

    /**
     * Accessor for an identifier for use in the datastore adapter
     * @param identifier The identifier name
     * @return Identifier name for use with the datastore adapter
     */
    String getIdentifierInAdapterCase(String identifier);

    /**
     * To be called when we want an identifier name creating based on the
     * identifier. Creates identifier for COLUMN, FOREIGN KEY, INDEX and TABLE
     * @param identifierType the type of identifier to be created
     * @param sqlIdentifier The SQL identifier name
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newIdentifier(int identifierType, String sqlIdentifier);

    /**
     * Method to use to generate an identifier for a datastore field with the
supplied name.
     * The passed name will not be changed (other than in its case) although it may
     * be truncated to fit the maximum length permitted for a datastore field
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
identifier.
     * @param identifierName The identifier name
     * @return The DatastoreIdentifier for the table
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(String identifierName);

    /**
     * Method to return a Table identifier for the specified class.
     * @param md Meta data for the class
     * @return The identifier for the table
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(AbstractClassMetaData md);

    /**
     * Method to return a Table identifier for the specified field.
     * @param fmd Meta data for the field
     * @return The identifier for the table
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(AbstractMemberMetaData fmd);

    /**
     * Method to use to generate an identifier for a datastore field with the
supplied name.
     * The passed name will not be changed (other than in its case) although it may
     * be truncated to fit the maximum length permitted for a datastore field
identifier.
     * @param identifierName The identifier name
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newDatastoreFieldIdentifier(String identifierName);

    /**
     * Method to create an identifier for a datastore field where we want the
     * name based on the supplied java name, and the field has a particular
     * role (and so could have its naming set according to the role).
     * @param javaName The java field name
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, Index ?
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newDatastoreFieldIdentifier(String javaName, boolean
embedded, int fieldRole);

    /**
     * Method to generate an identifier name for reference field, based on the
metadata for the
     * field, and the ClassMetaData for the implementation.
     * @param refMetaData the MetaData for the reference field
     * @param implMetaData the AbstractClassMetaData for this implementation
     * @param implIdentifier PK identifier for the implementation
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, collection
element ?
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newReferenceFieldIdentifier(AbstractMemberMetaData
refMetaData,
            AbstractClassMetaData implMetaData, DatastoreIdentifier implIdentifier,
boolean embedded, int fieldRole);

    /**
     * Method to return an identifier for a discriminator datastore field.
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     * @return The discriminator datastore field identifier
     */
    DatastoreIdentifier newDiscriminatorFieldIdentifier();

    /**
     * Method to return an identifier for a version datastore field.
     * @return The version datastore field identifier
     */
    DatastoreIdentifier newVersionFieldIdentifier();

    /**
     * Method to return a new Identifier based on the passed identifier, but adding
on the passed suffix
     * @param identifier The current identifier
     * @param suffix The suffix
     * @return The new identifier
     */
    DatastoreIdentifier newIdentifier(DatastoreIdentifier identifier, String
suffix);

    // RDBMS types of identifiers

    /**
     * Method to generate a join-table identifier. The identifier could be for a
foreign-key
     * to another table (if the destinationId is provided), or could be for a simple
column
     * in the join table.
     * @param ownerFmd MetaData for the owner field
     * @param relatedFmd MetaData for the related field (if bidirectional)
     * @param destinationId Identifier for the identity field of the destination
table
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, collection
element ?
     * @return The identifier.
     */
    DatastoreIdentifier newJoinTableFieldIdentifier(AbstractMemberMetaData ownerFmd,
AbstractMemberMetaData relatedFmd,
            DatastoreIdentifier destinationId, boolean embedded, int fieldRole);

    /**
     * Method to generate a FK/FK-index field identifier.
     * The identifier could be for the FK field itself, or for a related index for
the FK.
     * @param ownerFmd MetaData for the owner field
     * @param relatedFmd MetaData for the related field (if bidirectional)
     * @param destinationId Identifier for the identity field of the destination
table (if strict FK)
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g owner, index ?
     * @return The identifier
     */
    DatastoreIdentifier newForeignKeyFieldIdentifier(AbstractMemberMetaData
ownerFmd, AbstractMemberMetaData relatedFmd,
            DatastoreIdentifier destinationId, boolean embedded, int fieldRole);

    /**
     * Method to return an identifier for an index (ordering) datastore field.
     * @param mmd Metadata for the field/property that we require to add an
index(order) column for
     * @return The index datastore field identifier
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
     */
    DatastoreIdentifier newIndexFieldIdentifier(AbstractMemberMetaData mmd);

    /**
     * Method to return an identifier for an adapter index datastore field.
     * An "adapter index" is a column added to be part of a primary key when some
other
     * column cant perform that role.
     * @return The index datastore field identifier
     */
    DatastoreIdentifier newAdapterIndexFieldIdentifier();

    /**
     * Method to generate an identifier for a sequence using the passed name.
     * @param sequenceName the name of the sequence to use
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newSequenceIdentifier(String sequenceName);

    /**
     * Method to generate an identifier for a primary key.
     * @param table the table
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newPrimaryKeyIdentifier(DatastoreContainerObject table);

    /**
     * Method to generate an identifier for an index.
     * @param table the table
     * @param isUnique if the index is unique
     * @param seq the sequential number
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newIndexIdentifier(DatastoreContainerObject table, boolean
isUnique, int seq);

    /**
     * Method to generate an identifier for a candidate key.
     * @param table the table
     * @param seq Sequence number
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newCandidateKeyIdentifier(DatastoreContainerObject table,
int seq);

    /**
     * Method to create an identifier for a foreign key.
     * @param table the table
     * @param seq the sequential number
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newForeignKeyIdentifier(DatastoreContainerObject table, int
seq);
}
```

Be aware that you can extend *org.datanucleus.store.mapped.identifier.AbstractIdentifierFactory*.

Implementation

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Let's assume that you want to provide your own identifier factory *MyIdentifierFactory*.

```
package mydomain;

import org.datanucleus.store.mapped.identifier.AbstractSIdentifierFactory

public class MyIdentifierFactory extends AbstractIdentifierFactory
{
    /**
     * Constructor.
     * @param dba Datastore adapter
     * @param clr ClassLoader resolver
     * @param props Map of properties with String keys
     */
    public MyIdentifierFactory(DatastoreAdapter dba, ClassLoaderResolver clr, Map
props)
    {
        super(dba, clr, props);
        ...
    }

    .. (implement the rest of the interface)
}
```

Plugin Specification

When we have defined our "IdentifierFactory" we just need to make it into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_identifierfactory">
        <identifierfactory name="myfactory"
class-name="mydomain.MyIdentifierFactory"/>
    </extension>
</plugin>
```

Plugin Usage

The only thing remaining is to use your new *IdentifierFactory* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property **datanucleus.identifierFactory** to *myfactory* (the name you specified in the plugin.xml file).

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.8 SQL Methods

Plugins : RDBMS SQL Methods Support

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the support for JDOQL/JPQL methods in the "new" query mechanism. DataNucleus provides support for the majority of SQL methods that you are ever likely to need but is structured so that you could add on support for your own easily enough

The following sections describe how to create your own SQL Method plugin for DataNucleus.

Interface

Any SQL Method plugin will need to implement *org.datanucleus.store.rdbms.sql.method.SQLMethod* So you need to implement the following interface

```
import org.datanucleus.store.rdbms.sql.method;

public interface SQLMethod
{
    /**
     * Return the expression for this SQL function.
     * @param expr The expression that it is invoked on
     * @param args Arguments passed in
     * @return The SQL expression using the SQL function
     */
    public SQLExpression getExpression(SQLExpression expr, List args);
}
```

Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The expression on which the method is invoked. So if you have *{string}.myMethod(args)* then the first argument will be *{string}*
- The args are the arguments passed in to the method call. They will be SQLExpression/SQLLiteral.

So if we wanted to support *{String}.length()* as an example, so we define our class as

```
package mydomain;

import java.util.List;
import java.util.ArrayList;

import org.datanucleus.exceptions.NucleusException;
import org.datanucleus.store.rdbms.sql.expression.NumericExpression;
import org.datanucleus.store.rdbms.sql.expression.SQLExpression;
import org.datanucleus.store.rdbms.sql.expression.StringExpression;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class MyStringLengthMethod extends AbstractSQLMethod
{
    public SQLExpression getExpression(SQLExpression expr, List args)
    {
        if (expr instanceof StringExpression)
        {
            ArrayList funcArgs = new ArrayList();
            funcArgs.add(expr);
            return new NumericExpression("CHAR_LENGTH", funcArgs);
        }
        else
        {
            throw new NucleusException(LOCALISER.msg("060001", "length", expr));
        }
    }
}
```

So in this implementation when the user includes {*string*}.*length()* this is translated into the SQL **CHAR_LENGTH({string})** which will certainly work on some RDBMS. Obviously you could use this extension mechanism to support a different underlying SQL function.

Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_method">
        <sql-method class="java.lang.String" method="indexOf" datastore="h2"
            evaluator="mydomain.MyStringLengthMethod"/>
    </extension>
</plugin>
```

So we defined calls to a method *length* for the type *java.lang.String* for the datastore "h2" to use our evaluator. Simple! Whenever this method is encountered in a query from then on for the H2 database it will use our method evaluator.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.9 SQL Operations

Plugins : RDBMS SQL Operations Support

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the support for JDOQL/JPQL operations in the "new" query mechanism. DataNucleus provides support for all major operations typically handling them internally but occasionally handing off to an SQL function where one is more appropriate. However the codebase is structured so that you could add on support for your own easily enough

The following sections describe how to create your own SQL Operation plugin for DataNucleus.

Interface

Any SQL operation plugin will need to implement *org.datanucleus.store.rdbms.sql.operation.SQLOperation* So you need to implement the following interface

```
import org.datanucleus.store.rdbms.sql.method;

public interface SQLOperation
{
    /**
     * Return the expression for this SQL function.
     * @param expr Left hand expression
     * @param expr2 Right hand expression
     * @return The SQL expression for the operation
     */
    public SQLExpression getExpression(SQLExpression expr, SQLExpression expr2);
}
```

Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The expression on the left hand side of the operation. So if you have {*expr1*} {*operation*} {*expr2*} then the first argument will be {*expr1*}
- The expression on the right hand side of the operation. So if you have {*expr1*} {*operation*} {*expr2*} then the first argument will be {*expr2*}

So if we wanted to support *modulus (%)* (so something like **expr1 % expr2**) and wanted to use the SQL function "MOD" to provide this then we define our class as

```
package mydomain;

import java.util.ArrayList;

import org.datanucleus.exceptions.NucleusException;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
import org.datanucleus.store.rdbms.sql.operation.SQLOperation;
import org.datanucleus.store.rdbms.sql.expression.SQLExpression;
import org.datanucleus.store.rdbms.sql.expression.NumericExpression;

public class MyModOperation implements SQLOperation
{
    public SQLExpression getExpression(SQLExpression expr, SQLExpression expr2)
    {
        ArrayList args = new ArrayList();
        args.add(expr);
        args.add(expr2);
        return new NumericExpression("MOD", args);
    }
}
```

So in this implementation when the user includes *{expr1} % {expr2}* this is translated into the SQL **MOD({expr1}, {expr2})** which will certainly work on some RDBMS. Obviously you could use this extension mechanism to support a different underlying SQL function.

Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_operation">
        <sql-operation name="mod" datastore="hsql"
            evaluator="mydomain.MyModOperation"/>
    </extension>
</plugin>
```

So we defined calls to an operation *mod* for the datastore "hsql" to use our evaluator. Simple! Whenever this operation is encountered in a query from then on for the HSQL database it will use our operation evaluator.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.10 SQL Table Namer

Plugins : RDBMS SQL Table Naming Support

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the support for controlling the naming of table aliases in (some) SQL statements. DataNucleus provides a few options out of the box. The default is *alpha-scheme* which names tables based on the table-group they are in and the number in that group, so giving names like A0, A1, A2, B0, B1, C0, D0. It also provides a simpler option *t-scheme* that names all tables as "T{number}" so T0, T1, T2, T3, etc.

The following sections describe how to create your own SQL Table Namer plugin for DataNucleus.

Interface

Any SQL Table Namer plugin will need to implement *org.datanucleus.store.rdbms.sql.SQLTableNamer* So you need to implement the following interface

```
package org.datanucleus.store.rdbms.sql;

import org.datanucleus.store.rdbms.sql.SQLStatement;
import org.datanucleus.store.mapped.DatastoreContainerObject;

public interface SQLTableNamer
{
    /**
     * Method to return the alias to use for the specified table.
     * @param stmt The statement where we will use the table
     * @param table The table
     * @return The alias to use
     */
    public String getAliasForTable(SQLStatement stmt, DatastoreContainerObject
table);
}
```

Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The SQLStatement that is being constructed and that we create the names for
- The table that we want to generate the alias for

Lets just go through our default namer scheme to understand how it works

```
package mydomain;

import org.datanucleus.store.rdbms.sql.SQLTableNamer;
import org.datanucleus.store.rdbms.sql.SQLStatement;
import org.datanucleus.store.mapped.DatastoreContainerObject;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
public class MySQLTableNamer implements SQLTableNamer
{
    public String getAliasForTable(SQLStatement stmt, DatastoreContainerObject
table)
    {
        if (stmt.getPrimaryTable() == null)
        {
            return "T0";
        }
        else
        {
            return "T" + (stmt.getNumberOfTables() < 0 ? "1" :
(stmt.getNumberOfTables()+1));
        }
    }
}
```

So we simply name the primary table of the statement as "T0", and then all subsequent tables based on the number of the table. That was hard!

Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_tablenamer">
        <sql-tablenamer name="my-t-scheme" class="mydomain.MySQLTableNamer"/>
    </extension>
</plugin>
```

So now if we define a query using the extension *datanucleus.sqlTableNamingStrategy* set to "my-t-scheme" then it will use our table namer.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.11 RDBMS Requests

Plugins : RDBMS Requests

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the ability to define the code for a request operation. Request operations are INSERT, UPDATE, DELETE, FETCH and LOCATE. So you could write your own plugin to handle how fields are fetched from the datastore to make it more efficient for your case. DataNucleus provides default implementations for these operations, obviously.

Plugin extension-point	Name	Type	Description	Location
org.datanucleus.store.rdbms.request	default	insert	Insert of an object into a table	datanucleus-rdbms
org.datanucleus.store.rdbms.request	default	update	Update of an object into a table	datanucleus-rdbms
org.datanucleus.store.rdbms.request	default	delete	Delete of an object from a table	datanucleus-rdbms
org.datanucleus.store.rdbms.request	default	fetch	Fetch of an object from a table using SQLStatement	datanucleus-rdbms
org.datanucleus.store.rdbms.request	original	fetch	Fetch of an object from a table using original TJDO method	datanucleus-rdbms
org.datanucleus.store.rdbms.request	default	locate	Locate of an object from a table using SQLStatement	datanucleus-rdbms
org.datanucleus.store.rdbms.request	original	locate	Locate of an object from a table using original TJDO method	datanucleus-rdbms

The following sections describe how to create your own SQL Table Namer plugin for DataNucleus.

Interface

Any RDBMS Request plugin will need to extend *org.datanucleus.store.rdbms.request.Request* So you need to extend the following abstract class

```
public abstract class Request
{
    protected DatastoreClass table;
    protected PrimaryKey key;

    /**
     * Constructor, taking the table to use for the request.
     * @param table The Table to use for the request.
     **/
    public Request(DatastoreClass table)
    {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        this.table = table;
        this.key = ((AbstractClassTable)table).getPrimaryKey();
    }

    /**
     * Method to execute the request - to be implemented by deriving classes.
     * @param sm The StateManager for the object in question.
     **/
    public abstract void execute(StateManager sm);
}
```

Implementation

Obviously each request type will generate different types of SQL and have different input to the constructor. If you want to define your own, please refer to the default implementations for reference.

Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file *plugin.xml* to your JAR at the root. The file *plugin.xml* should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.rdbms_request">
        <rdbms-request name="my-fetch-request" type="fetch"
class="mydomain.MyFetchRequest"/>
    </extension>
</plugin>
```

and to enable its use you should set the persistence property **datanucleus.rdbms.request.fetch** to *my-fetch-request*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.1 Guides

DataNucleus Access Platform Guides

This section provides a series of worked examples of persistence. If you have any guide that you would like to contribute to be included here please contribute them via the forum.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.2 DataNucleus and Eclipse

DataNucleus and Eclipse

Eclipse provides a powerful development environment for Java systems. DataNucleus provides its own plugin for use within Eclipse, giving access to many features of DataNucleus from the convenience of your development environment.

- Installation
- General Preferences
- Preferences : Enhancer
- Preferences : SchemaTool
- Enable DataNucleus Support
- Generate JDO 2 MetaData
- Generate persistence.xml
- Run the Enhancer
- Run SchemaTool

Plugin Installation

The DataNucleus plugin requires Eclipse 3.1 or above. To obtain and install the DataNucleus Eclipse plugin select "Help -> *Software Updates* -> Find and Install". On the panel that pops up select "Search for new features to install". Select "New Remote Site", and in that new window set the URL as http://www.jpox.org/downloads/eclipse-update/ and the name as DataNucleus. Now select the site it has added "DataNucleus", and click "Finish". This will then find the releases of the DataNucleus plugin (and JPOX plguins). **Select the latest version of the DataNucleus Eclipse plugin**. Eclipse then downloads and installs the plugin. Easy!

Plugin configuration

The DataNucleus Eclipse plugin allows saving of preferences so that you get nice defaults for all subsequent usage. You can set the preferences at two levels :-

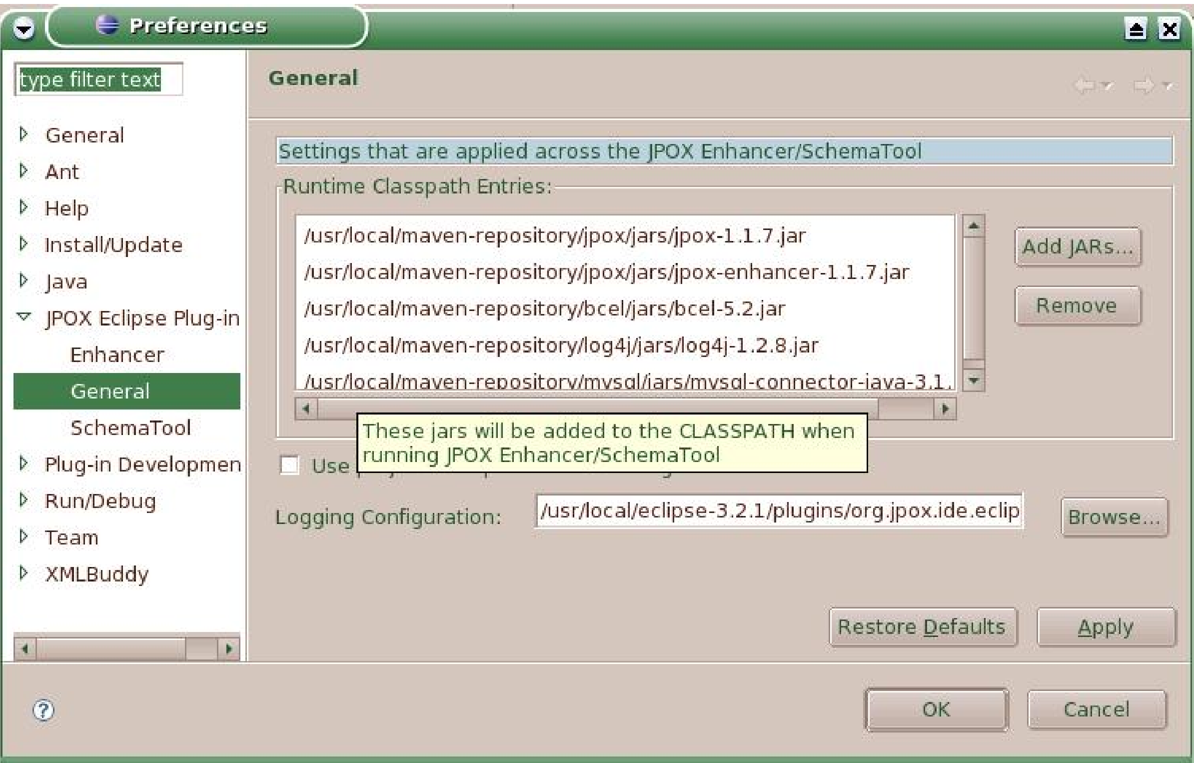
- **Globally for the Plugin** : Go to *Window* -> *Preferences* -> *DataNucleus Eclipse Plugin* and see the options below that
- **For a Project** : Go to *{your project}* -> *Properties* -> *DataNucleus Eclipse Plugin* and select "Enable project-specific properties"

Plugin configuration - General

Firstly open the main plugin preferences page and configure the libraries needed by DataNucleus. These are in addition to whatever you already have in your projects CLASSPATH, but to run the DataNucleus Enhancer you will require ASM/BCEL (depending which you are using), JDO (but you'll likely have this in your project CLASSPATH), DataNucleus Core/Enhancer/RDBMS jars, as well as LOG4J, and your

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

JDBC driver (if using RDBMS). Below this you can set the location of a configuration file for Log4j to use. This is useful when you want to debug the Enhancer/SchemaTool operations. Also make sure you identify the version of DataNucleus being used.

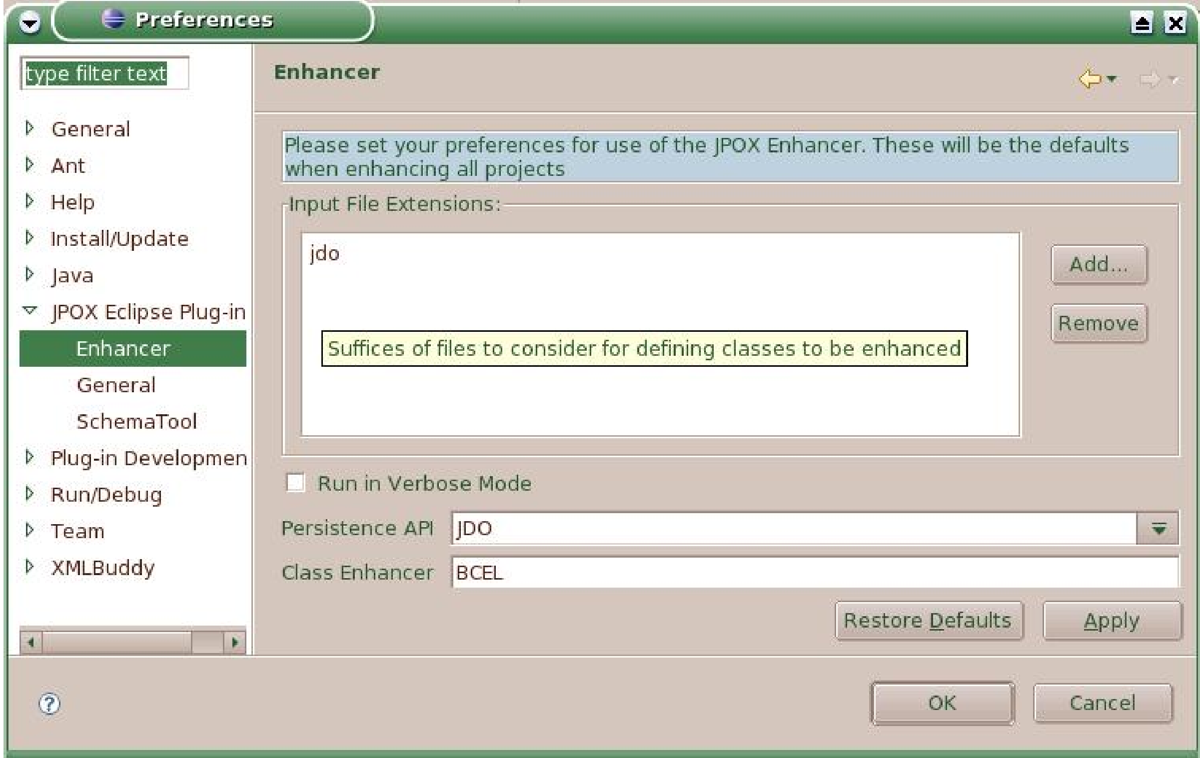


Plugin configuration - Enhancer

Open the "Enhancer" page. You have the following settings

- Input file extensions : the enhancer accepts input defining the classes to be enhanced. This is typically performed by passing in the JDO MetaData files. When you use annotations you need to pass in *class* files. So you select the suffices you need
- Verbose : selecting this means you get much more output from the enhancer
- Persistence API : DataNucleus supports JDO and JPA
- ClassEnhancer : DataNucleus provides enhancers using BCEL, or ASM. This allows you to select which one.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

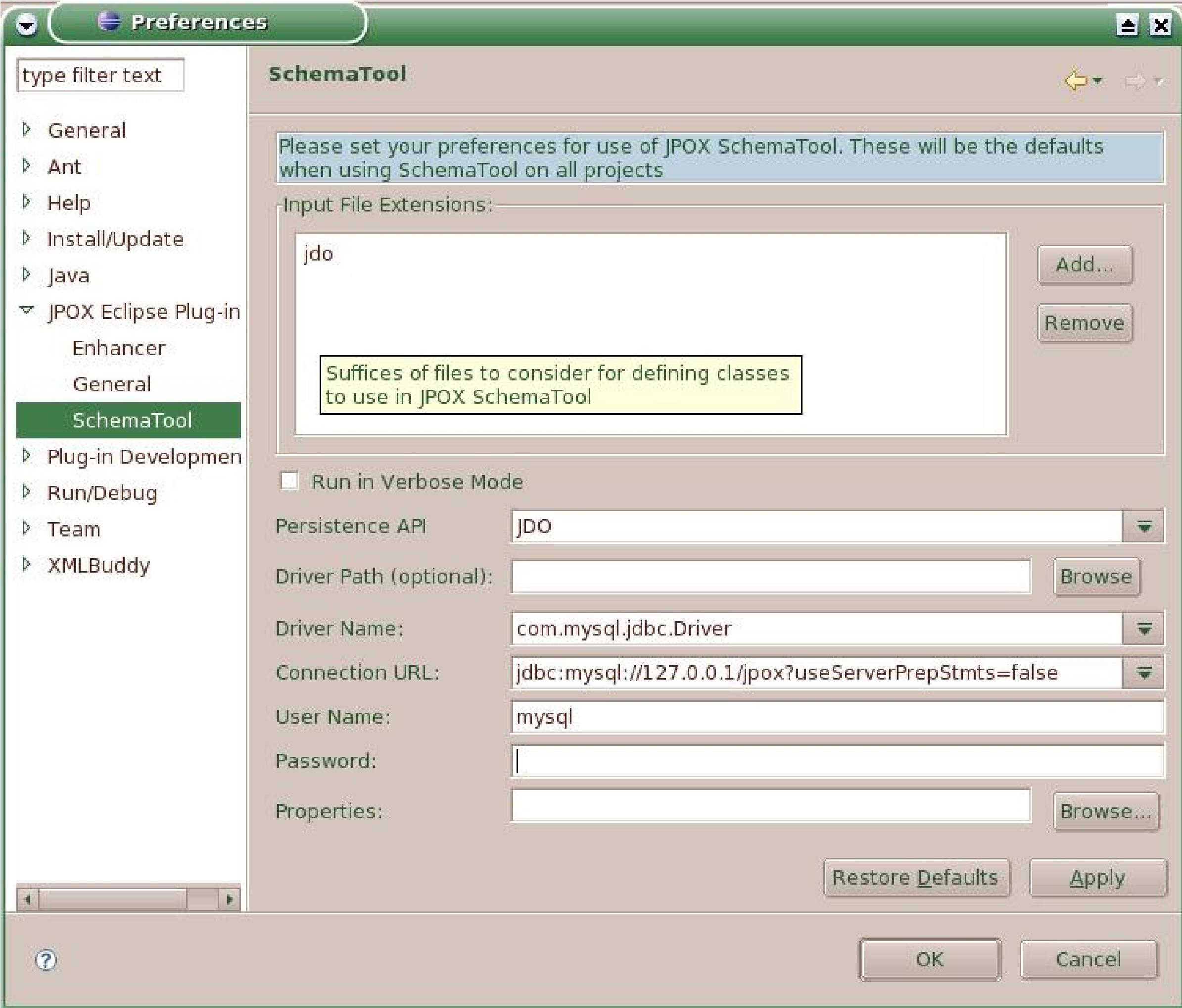


Plugin configuration - SchemaTool

Open the "SchemaTool" page. You have the following settings

- Input file extensions : SchemaTool accepts input defining the classes to have their schema generated. This is typically performed by passing in the JDO MetaData files. When you use annotations you need to pass in *class* files. So you select the suffices you need
- Verbose : selecting this means you get much more output from SchemaTool
- Persistence API : DataNucleus supports JDO and JPA
- Datastore details : You can either specify the location of a properties file defining the location of your datastore, or you supply the driver name, URL, username and password.

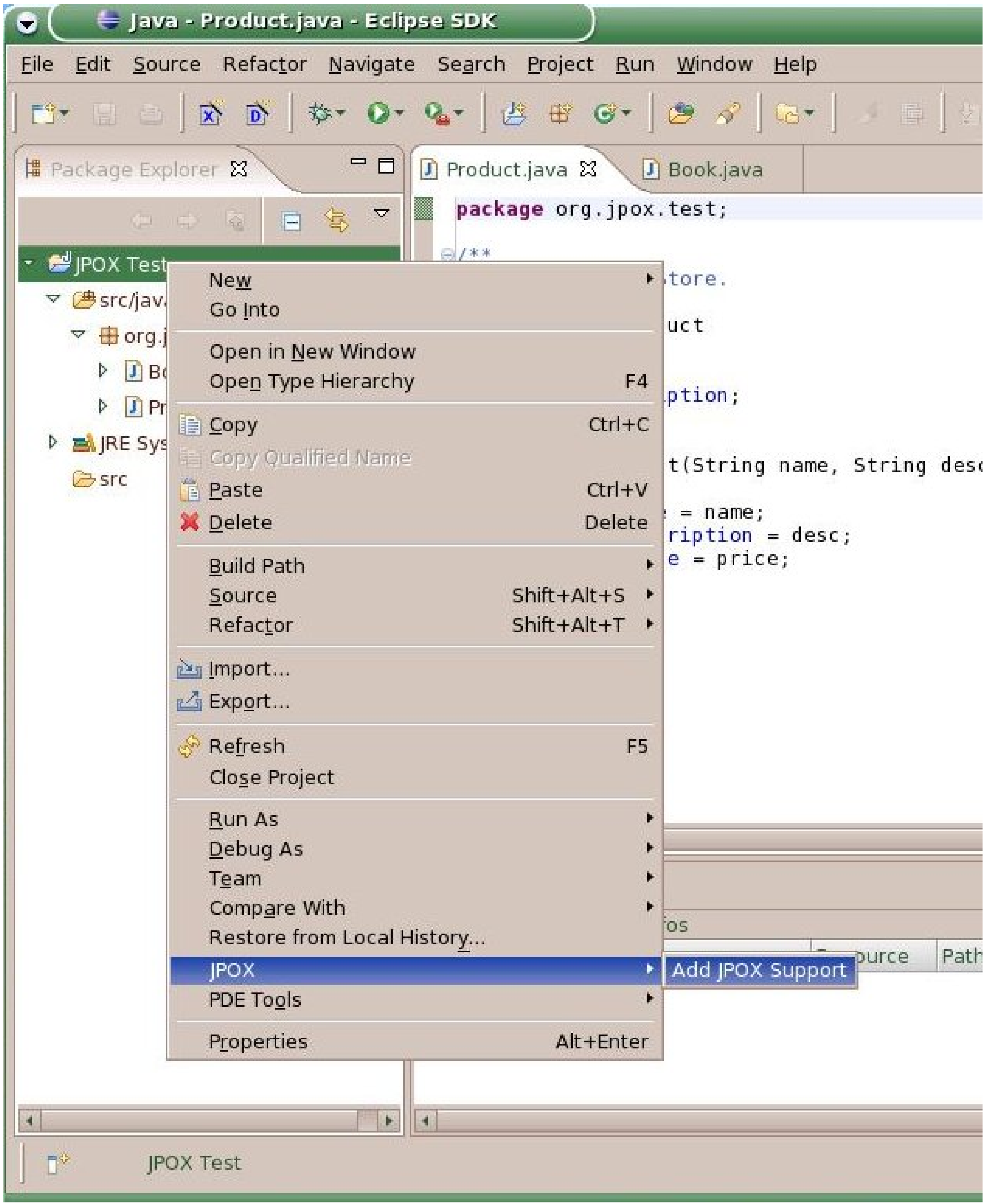
©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Enabling DataNucleus support

First thing to note is that the DataNucleus plugin is for Eclipse "Java project"s only. After having configured the plugin you can now add DataNucleus support on your projects. Simply right-click on your project and select (DataNucleus->"Add DataNucleus Support") from the context menu.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Defining JDO2 Metadata

It is standard practice to define the MetaData for your persistable classes in the same package as these classes. You now define your MetaData, by right-click on a package in your project and select "Create JDO 2.0 Metadata File" from DataNucleus context menu. The dialog prompts for the file name to be used and creates a basic Metadata file for all classes in this package, which can now be adapted to your

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

needs. You can also perform same steps as above on a *.java file, which will create the metadata for the selected file only. Please note that the wizard will overwrite existing files without further notice.

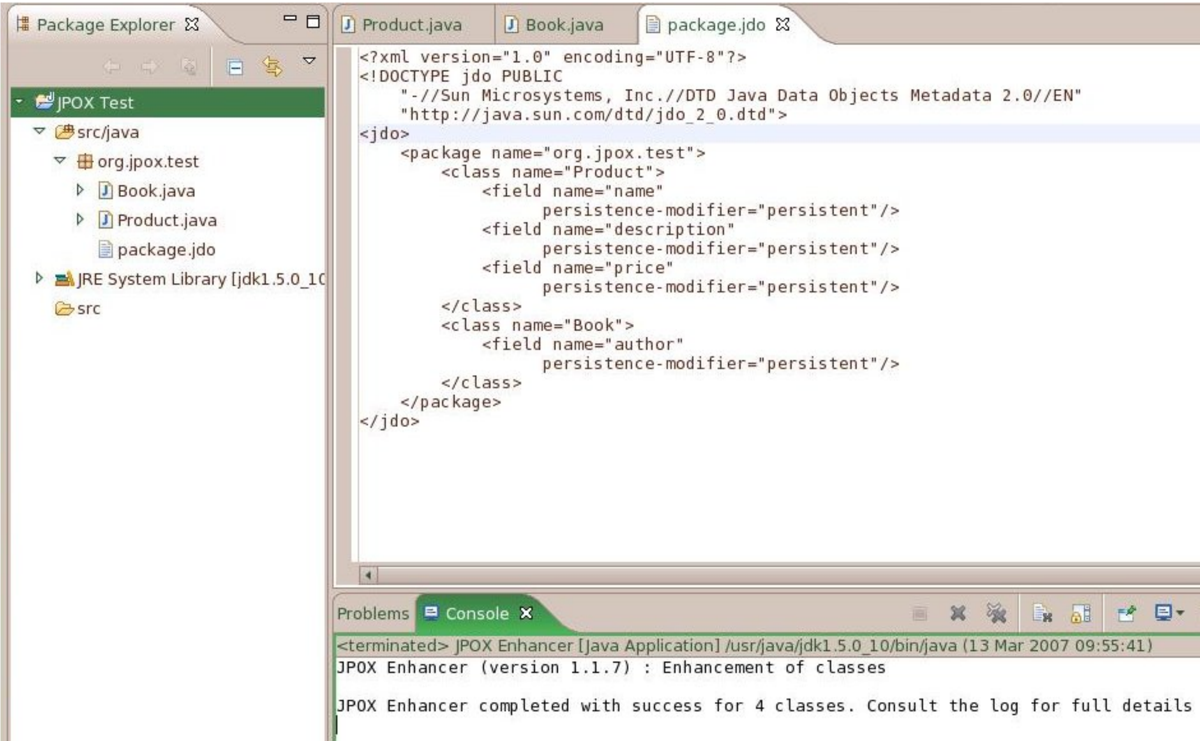
Defining 'persistence.xml'

You can also use the DataNucleus plugin to generate a "persistence.xml" file adding all classes into a single *persistence-unit*. You do this by right-clicking on your project, and selecting the option. The "persistence.xml" is generated under META-INF for the source folder. Please note that the wizard will overwrite existing files without further notice.

Enhancing the classes

The DataNucleus Eclipse plugin allows you to easily byte-code enhance your classes using the DataNucleus enhancer. Right-click on your project and select "Enable Auto-Enhancement" from the DataNucleus context menu. Now that you have the enhancer set up you can enable enhancement of your classes. The DataNucleus Eclipse plugin currently works by enabling/disabling automatic enhancement as a follow on process for the Eclipse build step. This means that when you enable it, every time Eclipse builds your classes it will then enhance the classes defined by the available "jdo" MetaData files. Thereafter every time that you build your classes the JDO enabled ones will be enhanced. Easy! Messages from the enhancement process will be written to the Eclipse Console. **Make sure that you have your Java files in a source folder, and that the binary class files are written elsewhere** If everything is set-up right, you should see the output below.

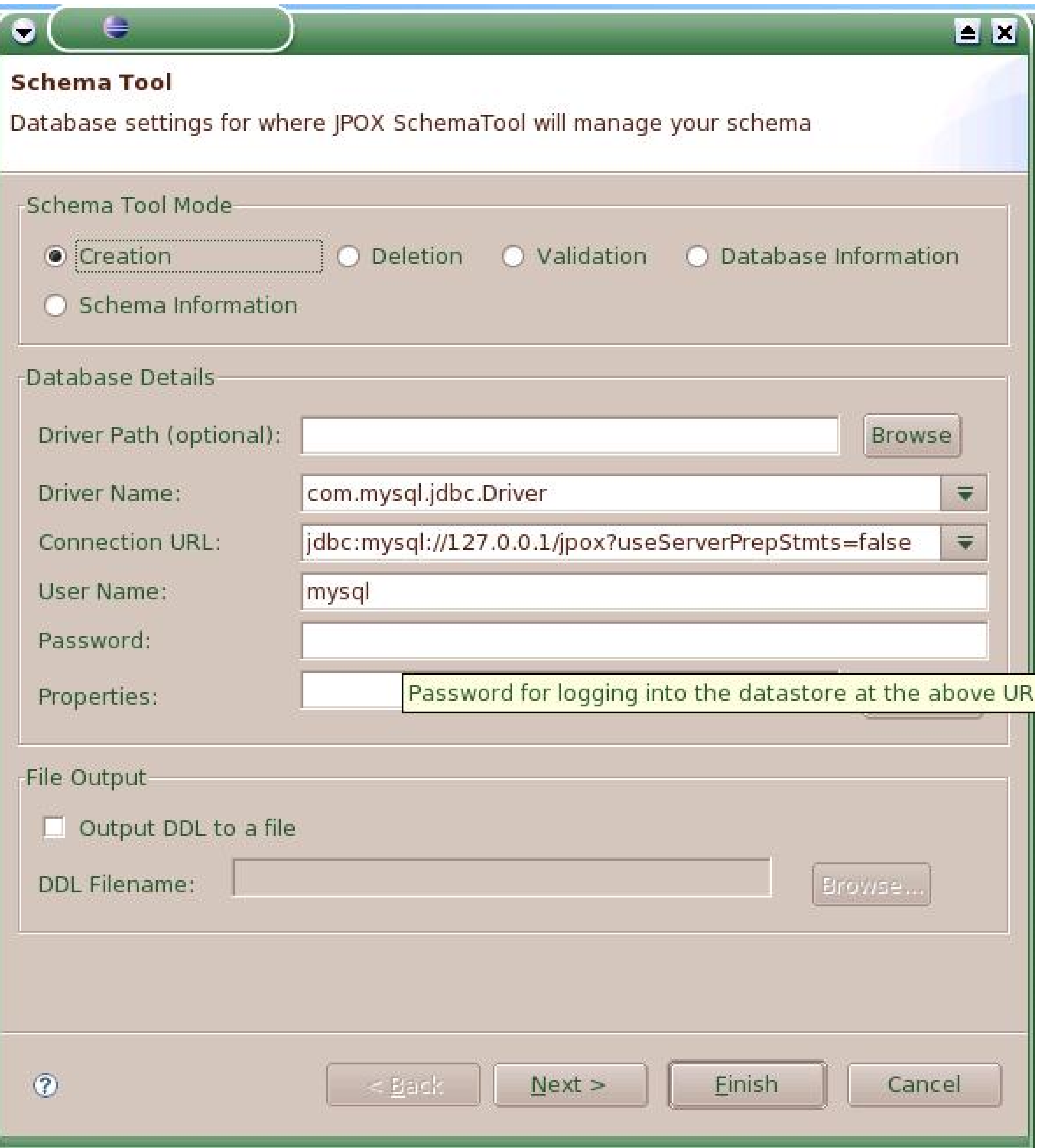
©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Generating your database schema

Once your classes have been enhanced you are in a position to create the database schema (assuming you will be using a new schema - omit this step if you already have your schema). Click on the project under "Package Explorer" and under "DataNucleus" there is an option "Run Schema Tool". This brings up a panel to define your database location (URL, login, password etc). You enter these details and the schema will be generated.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Messages from the SchemaTool process will be written to the Eclipse Console.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.3 DataNucleus and NetBeans

DataNucleus and NetBeans 4.0

Perhaps the most important step in developing applications with JPOX is the enhancement of compiled classes. NetBeans 4.x provides a convenient way of integrating this procedure into the build process without the need for any additional tool or plugin. This is possible because NetBeans 4.x stores project information in **.properties** files and relies on Ant for the build process. Class enhancement thus becomes a simple matter of adding a new task to the existing **build.xml** generated by NetBeans 4.x.

This tutorial shows how to integrate JPOX with NetBeans 4.x to simplify the development of JDO applications.

Requirements

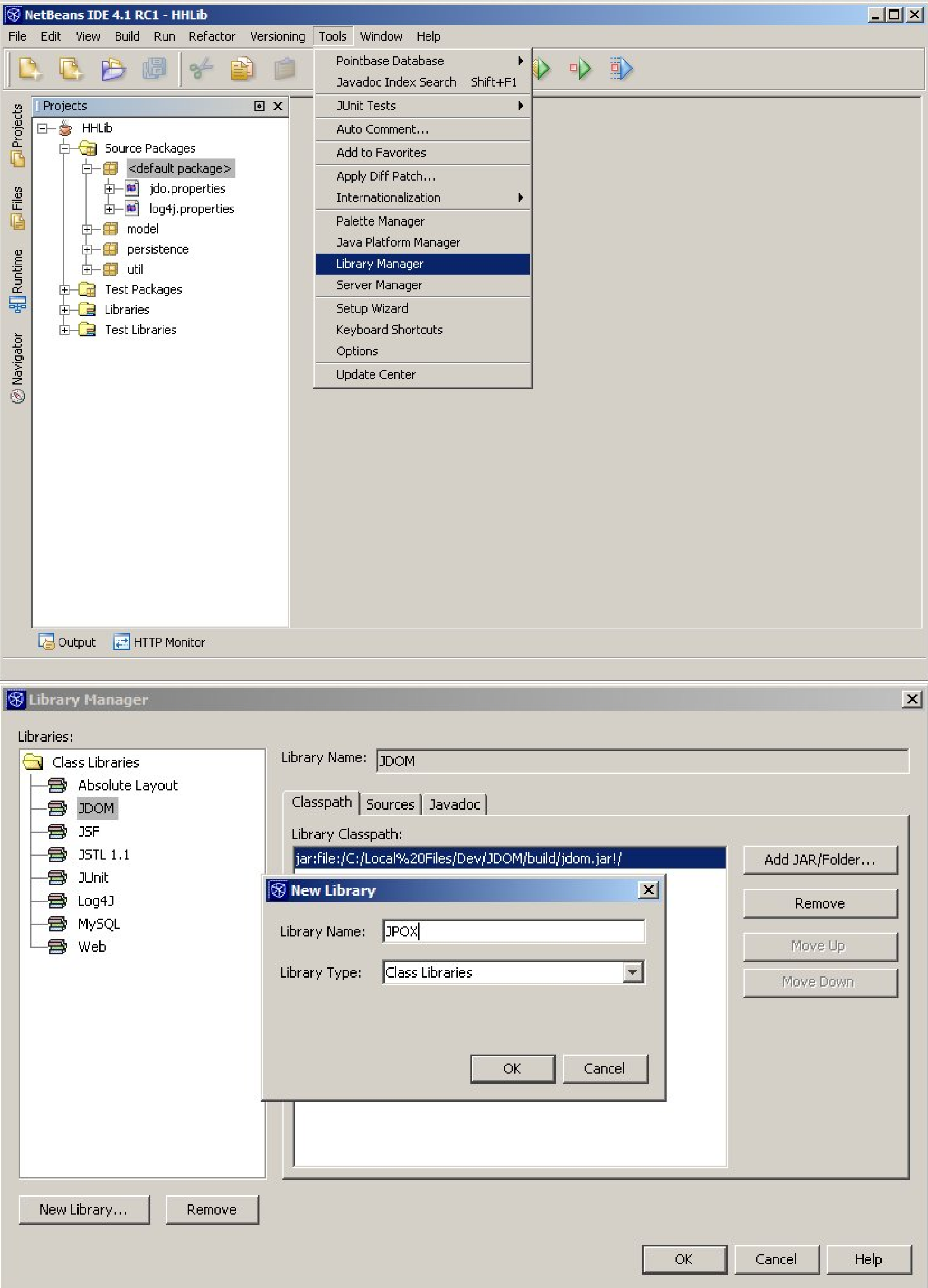
The following components are required to complete this tutorial successfully:

- JPOX "Core"
- JPOX Enhancer

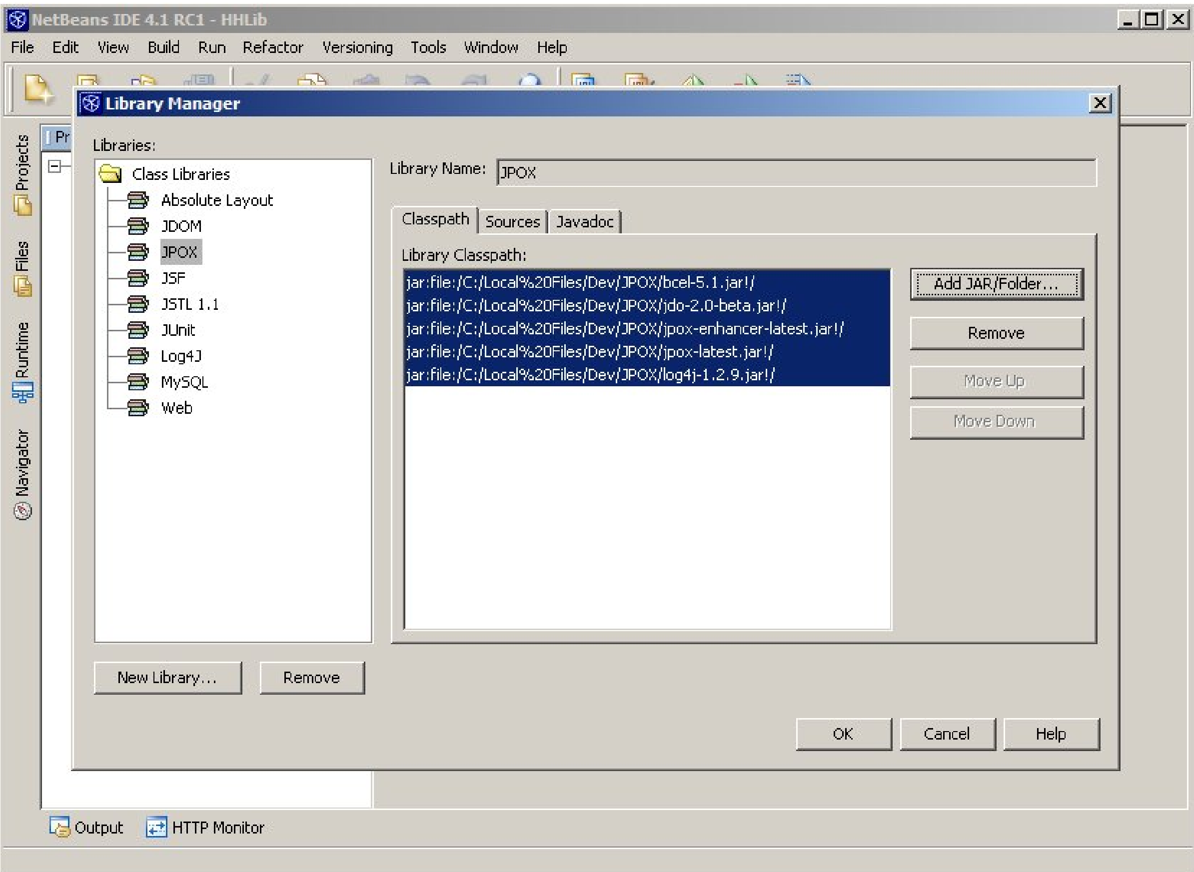
Setting up NetBeans 4.x for JPOX use

The first thing to do is to register the JPOX components in the **Library Manager** of NetBeans 4.x so that these become available to any project created with the IDE. This involves creating a new library and adding the JAR files to it, as shown in the following screenshots.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

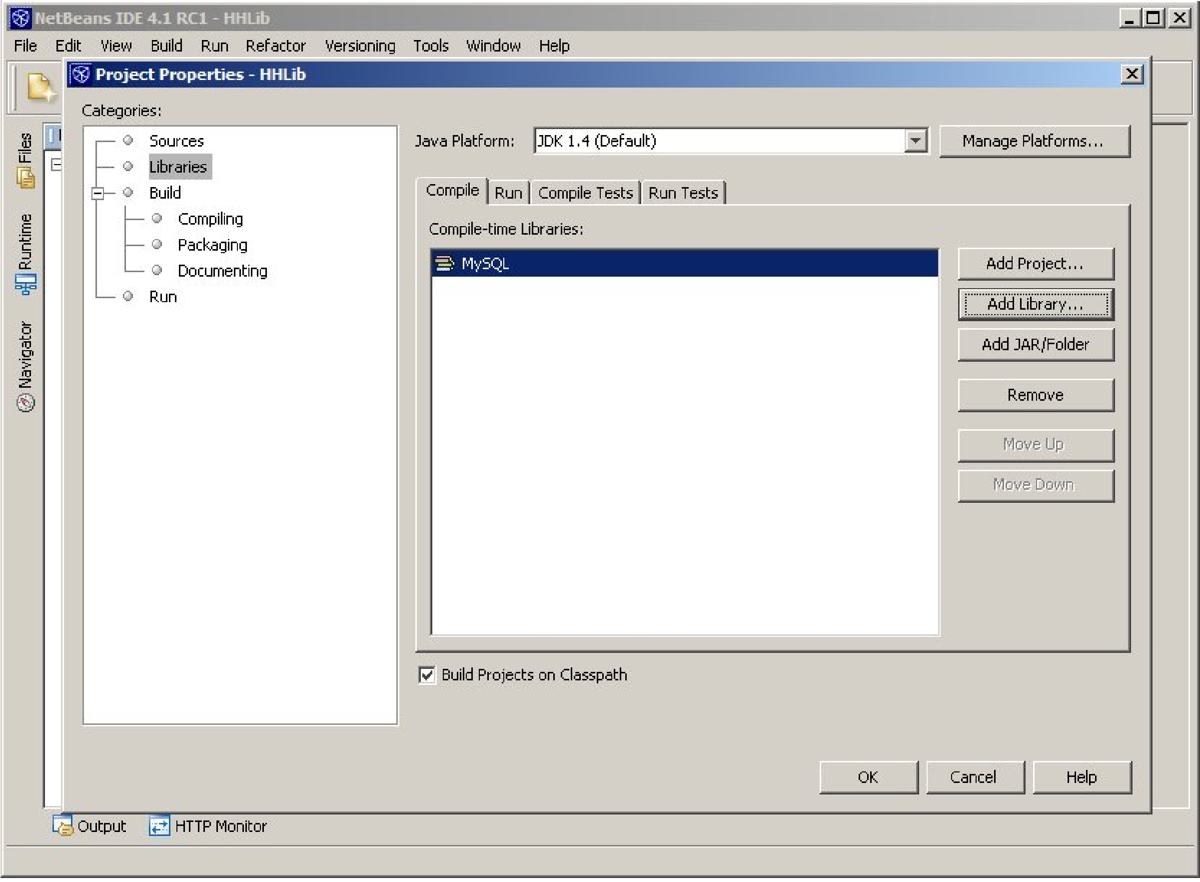


Once this is done, the NetBeans 4.x will add the JAR files to the classpath whenever the newly-created JPOX library is selected for a project.

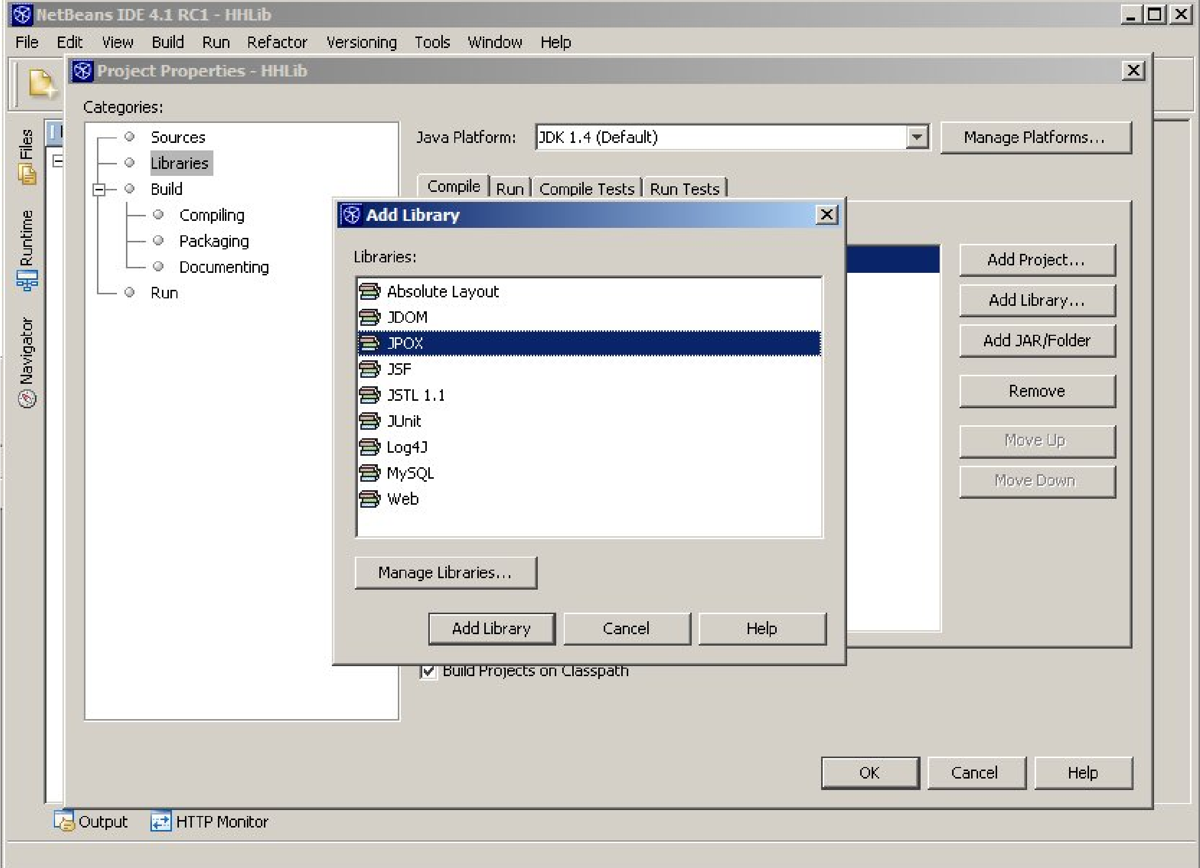
Setting up a new project

Now create a new project and add the JPOX library to it. This is done by viewing the properties for the project (right-click on the project and select *Properties...*). Click on *Add Library...* and select the JPOX library that we created in the previous section.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

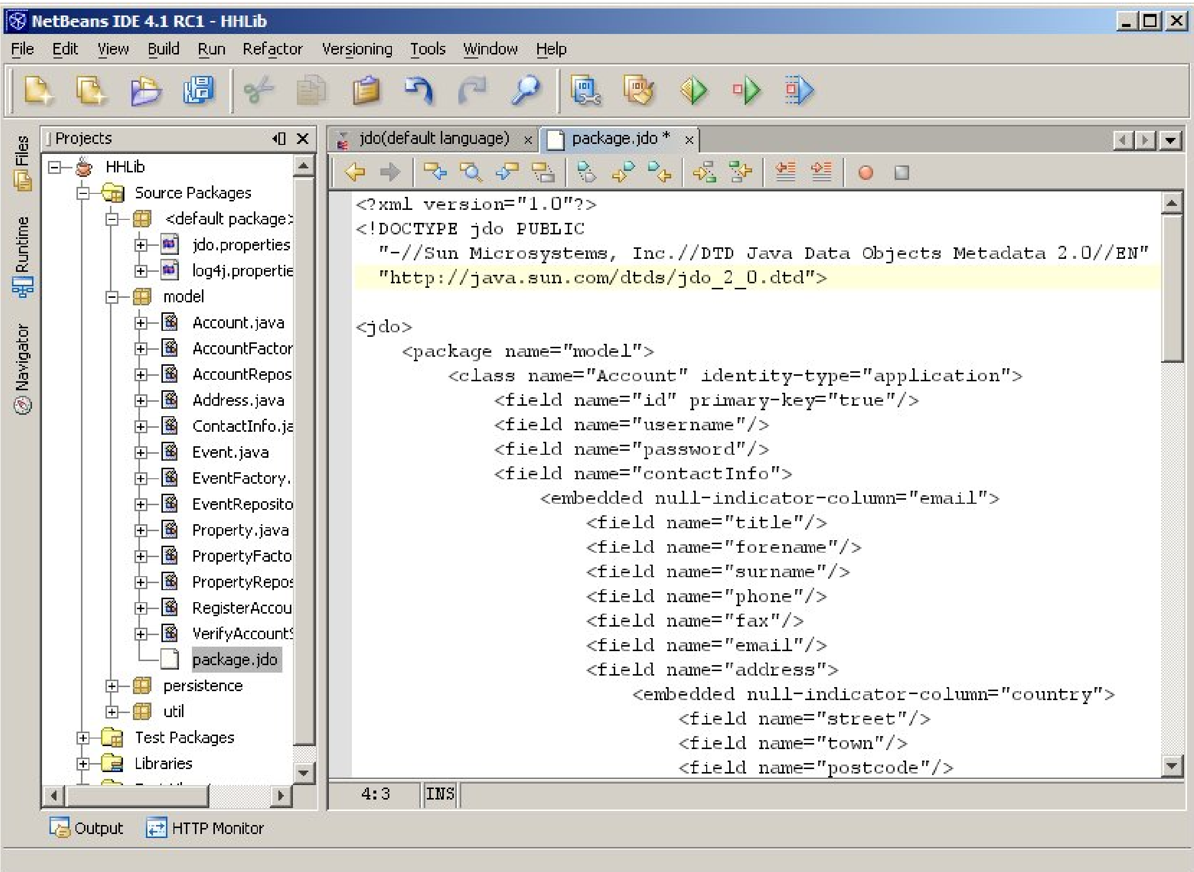


Note that I store the properties for the *PersistenceManagerFactory* and for Log4j in **.properties** files within the source root folder. This varies according to personal preferences, but I have found this to be very efficient.

Write code and the metadata

The next step is to write the code for the classes and the meta-data. This is a straightforward exercise left to the reader. One note of advice, however. Because NetBeans 4.x does not recognise the **package.jdo** file, it does not auto-complete XML code as it is being entered. The reader may wish to edit the file as **package.xml** and instruct Ant, through an entry in the **build.xml** file to rename that file to **package.jdo** just before the enhancement is performed. **This tutorial does not show how to do this.**

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Enhancing the classes

Once the code and the metadata has been written, the enhancement process needs to be defined and integrated into the build process. As stated in the introduction, this requires a simple change to the **build.xml** file.

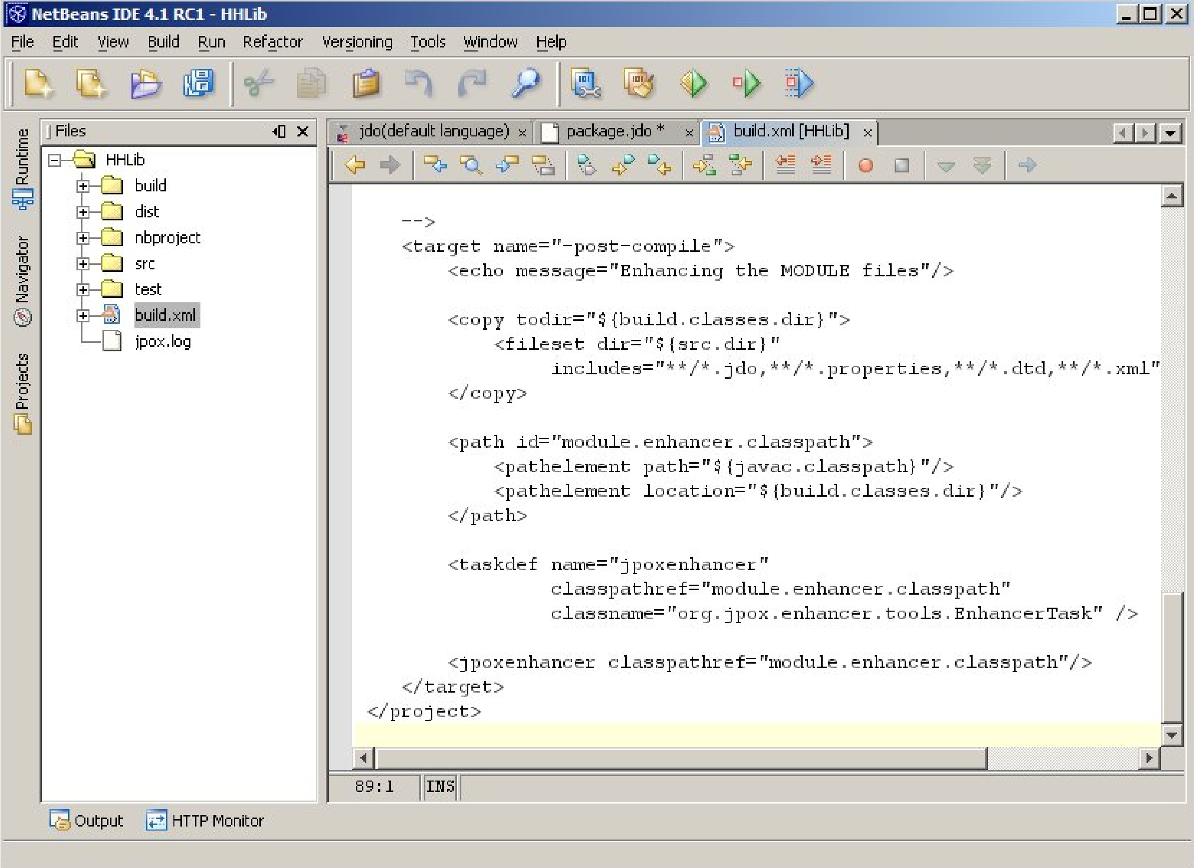
Click on the *Files* tab, expand the project tree, then open **build.xml**.

Override the **-post-compile** task/target with the following Ant instructions.

```
<target name="-post-compile">
    <echo message="Enhancing the MODULE files"/>

    <copy todir="${build.classes.dir}">
        <fileset dir="${src.dir}"
             includes="**/*.jdo,**/*.properties,**/*.dtd,**/*.xml"/>
    </copy>

    <path id="module.enhancer.classpath">
        <pathelement path="${javac.classpath}"/>
        <pathelement location="${build.classes.dir}"/>
    </path>

    <taskdef name="jpoxenhancer"
         classpathref="module.enhancer.classpath"
         classname="org.jpox.enhancer.tools.EnhancerTask" />
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <jpoxenhancer classpathref="module.enhancer.classpath"/>
    </target>
```

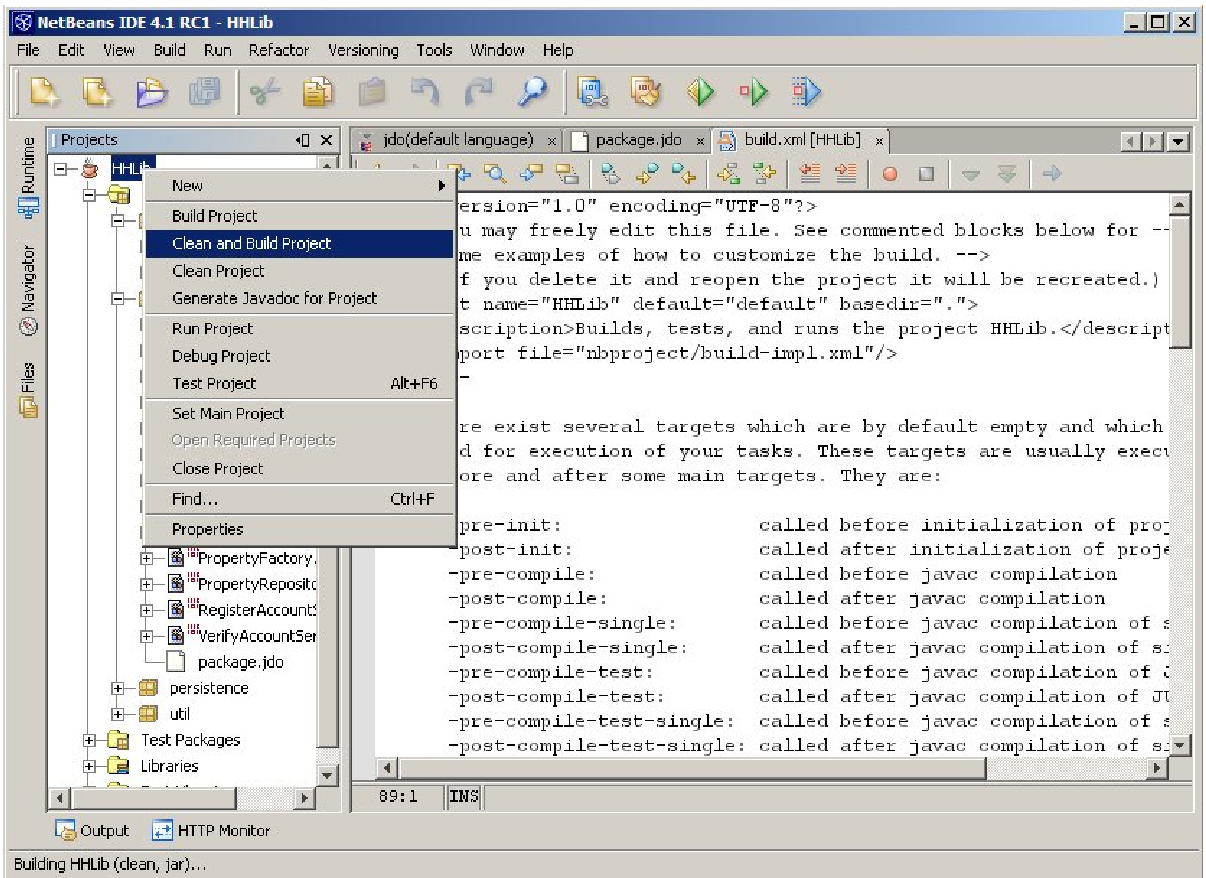


This target is the most convenient for enhancing classes because it occurs just after all classes have been compiled and is called in any case, whether the project is being built, tested or deployed. This ensures that classes are always enhanced.

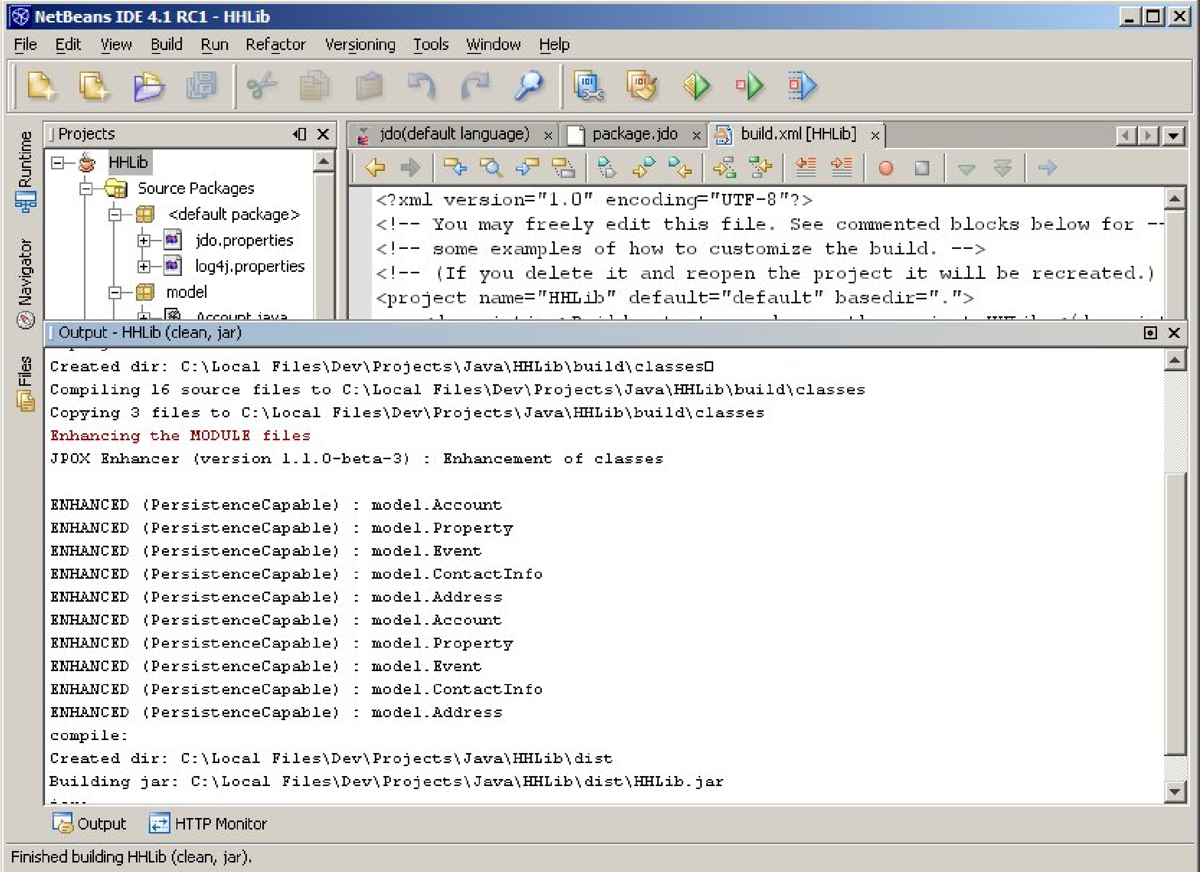
Building the project

The project can now be built, with the knowledge that the classes will be enhanced in the process.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Conclusion

This concludes our tutorial on how to integrate JPOX with NetBeans 4.x. As can be seen, thanks to NetBeans 4.x project system based on Ant, development of JDO applications is largely simplified. This tutorial was provided by a JPOX user Eddy Young.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.4 DataNucleus and Maven1

DataNucleus and Maven1

built by: maven

Apache Maven is a project management and build tool that is becoming more common in organisations. It continues on from where Ant left off and provides for modularisation of functionality into plugins, the use of a central repository for storing dependent JAR's, as well as many other convenience functionality. You can manage your project with Maven1 very easily. This tutorial describes how to create a simple Java project and how to use DataNucleus within that project. **Please note that DataNucleus supports use of Maven1 and Maven2. This guide reflects the support for Maven1 only**

Create the Project

The first step is to create the project itself. The Maven website should be used as the main reference for this step, but we cover here the essential parts of that step. Firstly, to start a project you need to establish a file structure to use. Maven allows you to configure this to your own tastes. In our example we create a file structure as follows

```
project.xml
project.properties
maven.xml
log4j.properties
src/java/... (source code)
src/test/... (unit tests)
xdocs/... (documentation)
```

As you see we have our source code in a single tree under src/java. Since, in our example, we adopt the practice of Test Driven Development (TDD) we also have a source tree for our unit tests (using JUnit) in src/test. We provide our documentation in a directory xdocs. In addition there are several files at the top level to control the operation of the project.

project.xml

This file provides the overall definition of our project. Here we present our file for this example

```
<?xml version="1.0"?>
<project>
    <pomVersion>3</pomVersion>
    <name>DataNucleus-Maven</name>
    <id>datanucleus-samples-maven</id>
    <currentVersion>1.1</currentVersion>
    <inceptionYear>2004</inceptionYear>
    <package>org.datanucleus.samples.maven</package>
    <shortDescription>DataNucleus Samples : Maven</shortDescription>
    <description>Demonstration of a Maven controlled project with
DataNucleus</description>
    <siteAddress>www.datanucleus.org</siteAddress>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <!-- Developers/Contributors for this project -->
    <developers>
        <developer>
            <name>Andy Jefferson</name>
            <roles>
                <role>Developer</role>
            </roles>
        </developer>
    </developers>

    <!-- Software Dependencies -->
    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.8</version>
            <url>http://jakarta.apache.org/log4j/</url>
        </dependency>

        <dependency>
            <groupId>javax.jdo</groupId>
            <artifactId>jdo2-api</artifactId>
            <version>2.1-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>mysql-connector-java</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>3.1.12</version>
        </dependency>
    </dependencies>

    <!-- Build process -->
    <build>
        <sourceDirectory>src/java</sourceDirectory>
        <unitTestSourceDirectory>src/test</unitTestSourceDirectory>
    </build>
</project>
```

So you see that in the header we have defined our project's package naming and the current version. We then define the developers for the project. There follows a crucial part which defines the software dependencies of our project. Finally we define where our source and test code is stored. This has the effect of allowing Maven to find things when it runs its numerous plugins and means that the tasks of creating JAR's, WAR's , and EAR's is trivial.

project.properties

This file allows control of the Maven plugins operation for this project. We will add things to this file later.

maven.xml

This file is used if we want to automate the calling of particular things when developing our system. This will be updated later.

Maven Repository

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The vast majority of software has dependencies. Any project that uses DataNucleus will require the DataNucleus JAR(s) to operate and so DataNucleus is a dependency of that project. Many projects adopt a policy of having a lib directory at the top level and putting all JAR's in that directory. This usually leads to the developer having multiple copies of the same JAR on their machine. Maven provides the idea of a centralised repository where these JAR's are stored. When Maven comes across a dependency it will first look in its repository for the JAR. If it doesn't find the JAR in the repository it can search for the JAR in a remote repository. The user can control where Maven searches for this repository. By default it will look at IBiblio. The location of these repositories can be defined by a file build.properties in the users home directory, and you specify the locations like this

```
maven.repo.local=/usr/local/maven-repository
maven.repo.remote=http://www.ibiblio.org/maven/
```

DataNucleus's website provides a list of dependencies that are required for using DataNucleus in your system. Maven will take care of dependencies like BCEL and Log4J. Some dependencies will however need to be downloaded from SUN's website (things like JTA, JAAS if you are using the parts of DataNucleus that requires these). You then need to manually copy these into your Maven Repository. For example, if you have your Maven repository at /usr/local/maven-repository and you want to put *transaction-api-1.1.jar* into the repo, you would create directories and copy the file so it looks like this

```
/usr/local/maven-repository/
/usr/local/maven-repository/javax.transaction/
/usr/local/maven-repository/javax.transaction/jars/
/usr/local/maven-repository/javax.transaction/jars/transaction-api-1.1.jar
```

Writing your classes

Clearly you have to write your system at some point in the process, and we don't tell you how to do that here ;-). The only thing you need to do is place your Java files under the source directory src/java, and your unit tests under src/test. Whilst writing your classes, it is advised that you structure your project in such a way that the model/domain classes are in their own area. This means that you can easily separate your data persistence layer from your business logic, or web tier, or whatever. When you identify which classes are to be persisted, you write the JDO MetaData file(s) for these classes, and put it in the same directory.

Building your project

As mentioned earlier, Maven builds your project using the locations that are defined in the project.xml and using its plugins. To build a JAR of your system you simply type

```
maven jar
```

How simple was that ? This has generated the class files under target/classes and has created the JAR file under target. Similarly to clean out the previous class files you do

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
maven clean
```

To clean and then build you would do

```
maven clean jar
```

To just compile the classes

```
maven java:compile
```

To run the unit tests you would do

```
maven test
```

Each of these 'clean', 'jar', 'test' are known as goals. Each plugin provides many goals.

Using DataNucleus with your project

As shown earlier we have defined DataNucleus as a dependency of our project. This means that it will include the DataNucleus JAR in the CLASSPATH when it compiles, and runs the unit tests, etc. DataNucleus provides its own Maven plugin to provide enhancement and generation of the schema. You should install this plugin into your $MAVEN_HOME/plugins directory.

The next thing that you will require is the Apache JDO JAR. This can be obtained from the DataNucleus Download page. When you have this JAR you need to install it manually into your Maven repository.

The final thing is to download the DataNucleus Maven plugin. This can be downloaded from the same page as the previous download. You then install this into your Maven installation. By this I mean, install it under $MAVEN_HOME/plugins/. You are now ready to go

Enhancing Classes

Once you have a set of compiled Java classes in target/classes you can enhance them for use with DataNucleus. The first thing we do here is to define some plugin properties in our project.properties. Here we add the lines

```
maven.datanucleus.jdo.fileset.dir=${basedir}/src/java
maven.datanucleus.log4j.configuration=file:${basedir}/log4j.properties
maven.datanucleus.verbose=true
```

These settings will use our log4j.properties file for log definitions and will use verbose mode for output. So let's enhance our classes.

```
maven datanucleus:enhance
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Can't get much simpler than that.

Generating the Schema

Let's assume that we are creating our own database schema and we want to use SchemaTool. We can do this also using Maven. Lets add some more properties to the project.properties

```
maven.datanucleus.database.driver=com.mysql.jdbc.Driver
maven.datanucleus.database.url=jdbc:mysql://localhost/datanucleus
maven.datanucleus.database.user=mysql
maven.datanucleus.database.password=
```

Change these settings to match your preferred RDBMS and database location. In this example above we are going to create the schema in a MySQL database called "datanucleus" running on our local machine. So that only leaves us to create the schema, which we do as follows

```
maven datanucleus:schema-create
```

This creates our tables and we're ready to run our system!.

Automated Enhancement

As you saw above we can enhance our classes very easily. We can however make this automatic after we do any compilation. To do this we need to add something to the file maven.xml

```
<project xmlns:maven="jelly:maven" xmlns:ant="jelly:ant"
    xmlns:j="jelly:core" xmlns:u="jelly:util">

    <!-- Perform enhance directly after compilation -->
    <postGoal name="java:compile">
        <copy todir="${basedir}/target/classes">
            <fileset dir="${basedir}/src/java">
                <include name="**/*.jdo"/>
            </fileset>
        </copy>
        <attainGoal name="datanucleus:enhance"/>
    </postGoal>

</project>
```

This code above will mean that after performing the java:compile goal (a compilation), Maven will copy the JDO MetaData files across into the target/classes area and will then run the datanucleus:enhance goal

Conclusion

Here you've seen how easy it is to integrate DataNucleus into the Maven build system. If you have any questions about Maven, then please ask them on the Maven site. Any questions about the Maven DataNucleus plugin should be asked on the DataNucleus Forum

If you want to get the latest DataNucleus, we have our own Maven repository containing the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

latest and greatest DataNucleus builds. These can be accessed by specifying a Maven repository of "http://www.datanucleus.org/downloads/maven/"

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.5 DataNucleus and Maven2

DataNucleus and Maven2

Apache Maven is a project management and build tool that is quite common in organisations. It continues on from where Maven1 left off though not providing backwards compatibility with Maven1.

Repository

The main thing that you need to bear in mind is that DataNucleus jars are present in DataNucleus own Maven repository so all you need to do is point your Maven2 project at this repository. You do this by including the following in your *pom.xml*

```
<project>
    ...


    <repositories>
        <repository>
            <id>DataNucleus_Repos</id>
            <name>DataNucleus Repository</name>
            <url>http://www.datanucleus.org/downloads/maven</url>
            <layout>legacy</layout>
        </repository>
        <repository>
            <id>DataNucleus_Repos2</id>
            <name>DataNucleus Repository</name>
            <url>http://www.datanucleus.org/downloads/maven2</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>DataNucleus_2</id>
            <url>http://www.datanucleus.org/downloads/maven2/</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

So this make the DataNucleus Maven1 and Maven2 repositories available, and also makes the DataNucleus Maven2 plugin available.

Maven2 Plugin

Now that you have the DataNucleus jars available to you, via the repositories, you want to perform DataNucleus operations. The primary operations are enhancement and SchemaTool (for RDBMS). To run the enhancer you do

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
mvn datanucleus:enhance
```

If you want this to be done automatically after compiling then you should add the following into your pom.xml

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.datanucleus</groupId>
                <artifactId>maven-datanucleus-plugin</artifactId>
                <version>1.0-SNAPSHOT</version>
                <configuration>
                    <mappingIncludes>**/*.jdo, **/*.class</mappingIncludes>
<log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
                    <verbose>true</verbose>
                    <enhancerName>ASM</enhancerName>
                    <props>${basedir}/datanucleus.properties</props>
                </configuration>
                <executions>
                    <execution>
                        <phase>compile</phase>
                        <goals>
                            <goal>enhance</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugin>
    </build>
</project>
```

So whenever compile takes place it then does an enhance using the DataNucleus Maven2 plugin.

SchemaTool is achieved similarly, via

```
mvn datanucleus:schema-create
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.6 DataNucleus and Ivy

DataNucleus with Ant and Ivy

Background

This guide introduces Apache Ant Ivy as tool to manage project dependencies for DataNucleus based applications.

In this quick tutorial you will learn how to use Apache Ant Ivy to compile and enhance persistent classes.

Enhancement

DataNucleus enhances persistent classes during compile time. It's a transparent process when using the JDK 1.6 or upper, otherwise use the DataNucleus Ant Enhancer Task (See Ant Guide).

Build files

There are two ivy files and one traditional ant build file:

- ivy.xml
- ivysettings.xml
- build.xml

The *ivy.xml* declares the project dependencies to DataNucleus libraries and the *ivysettings.xml* declares the DataNucleus SNAPSHOT repository.

The *ivy.xml* file:

```
<ivy-module version="2.0">
        <info organisation="datanuleus" module="sample" revision="1.0.0" />
        <configurations>
                <conf name="build" description="build dependancies" />
                <conf name="enhancer" extends="build" description="enhancer
dependances" />
                <conf name="runtime" extends="build" description="runtime
dependances" />
        </configurations>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <publications>
                <artifact />
        </publications>
        <dependencies>
                <dependency org="org.datanucleus" name="datanucleus-core"
rev="1.1-SNAPSHOT" conf="build->default"/>
                <dependency org="org.datanucleus" name="datanucleus-enhancer"
rev="1.1-SNAPSHOT" conf="enhancer->default"/>
                <dependency org="asm" name="asm" rev="3.0"
conf="enhancer->default"/>
                <dependency org="javax.jdo" name="jdo2-api" rev="2.3-SNAPSHOT"
conf="build->default"/>
                <dependency org="org.datanucleus" name="datanucleus-rdbms"
rev="1.1-SNAPSHOT" conf="runtime->default"/>
        </dependencies>

</ivy-module>
```

The *ivysettings.xml* file:

```
<ivysettings>
  <settings defaultResolver="resolver"/>
  <resolvers>
    <chain name="resolver">
      <url name="datanucleus-nightly" m2compatible="true">
        <artifact
pattern="http://www.datanucleus.org/downloads/maven2-nightly/[organisation]/[module]/[revision]/[artifact
/>
      </url>
      <ibiblio name="ibiblio" m2compatible="true"/>
      <filesystem name="repository">
        <ivy
pattern="${ivy.settings.dir}/../repository/[organisation]/[module]/[module]-[revision].xml"
/>
        <artifact
pattern="${ivy.settings.dir}/../repository/[organisation]/[artifact]/[artifact]-[revision].[ext]"
/>
      </filesystem>
    </chain>
  </resolvers>
</ivysettings>
```

The *build.xml* file:

```
<project basedir="." default="default" name="build"
xmlns:ivy="antlib:org.apache.ivy.ant">
    <property environment="env"/>
    <property name="debuglevel" value="source,lines,vars"/>
    <property name="target" value="1.5"/>
    <property name="source" value="1.5"/>
    <property name="bin" value="bin"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <property name="src" value="src"/>
    <property name="dist" value="target"/>

        <target name="default" depends="retrieve,compile,jar,publish"></target>

    <target name="retrieve" description="retrieve dependencies with ivy">
        <ivy:retrieve/>
            <ivy:cachepath pathid="build.path" conf="build" />
            <ivy:cachepath pathid="enhancer.path" conf="enhancer" />
    </target>
    <target name="publish">
            <ivy:cleancache />

        <ivy:publish deliverivypattern="ivy.xml" resolver="repository"
overwrite="true">
                <artifacts pattern="${dist}/[artifact]-[revision].[ext]"/>
        </ivy:publish>
    </target>
        <target name="compile">
                <delete dir="${bin}" failonerror="true"></delete>
                <mkdir dir="${bin}"/>
        <copy todir="${bin}">
            <fileset dir="${src}" includes="**/**.jdo" />
        </copy>
                <!-- using JDK 1.6 and datanucleus-core+datanucleus-enhancer+asm
will enhance the classes automatically -->
                <!-- MUST use fork, so datanucleus ensures that classes are enhanced
before going ahead with the build -->
                <javac srcdir="${src}"
                       destdir="${bin}"
                       source="${source}"
                       target="${target}"
                       debuglevel="${debuglevel}"
                       debug="true"
                       fork="true">
                        <classpath refid="build.path"></classpath>
                        <classpath refid="enhancer.path"></classpath>
                </javac>
        </target>

        <target name="jar">
                <delete dir="${dist}" failonerror="false"></delete>
                <mkdir dir="${dist}"/>
        <jar destfile="${dist}/${ivy.module}-${ivy.revision}.jar">
            <fileset dir="${src}" includes="**/**.jdo" />
            <fileset dir="${bin}"/>
        </jar>
        </target>

</project>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.7 Enhancing with Ant

Enhancing with Ant

Background

Enhancement is a JDO standard process to inject code into the persistent class. The injected code provides the basis of the solid and transparent object persistence specified by JDO.

In this quick tutorial you will learn how to use Apache Ant to compile and enhance persistent classes.

Enhancement

DataNucleus provides an Ant task to enhance files and all you have to do is set up some parameters, but before enhancing files with the DataNucleus Enhancer for Ant, let's look at a complete Ant build file specially created to compile, enhance and generate archive files (jar, war).

Enhancer Task

The Ant build file provided here as example is divided in many targets as like clean, compile, enhance, distribution and others. The one we are interested is the *enhance*.

This first section of the target *enhance* defines the classpath to use for enhancement. In this classpath we put the *DataNucleus jar* + *DataNucleus-Enhancer jar* + *DataNucleus dependencies* + *Your Persistent Classes (.class)* + *Your JDO Classes (.jdo)* + *Your dependencies*.

```
<!-- the classpath to enhance -->
<path id="module.enhancer.classpath">
    <pathelement location="${module.classes.dir}"/>
    <path refid="module.lib.classpath"/>
</path>
```

In the second section, we define the DataNucleus Enhancer task.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<!-- define the task enhancer -->
<taskdef name="nucleusenhancer"
    classpathref="module.enhancer.classpath"
    classname="org.datanucleus.enhancer.tools.EnhancerTask" />
```

With the DataNucleus Enhancer task defined, we are now able to use it. It's important to correctly define the classpath. See this reference for further information on attributes that can be configured for the enhancer.

```
<!-- enhance -->
<nucleusenhancer classpathref="module.enhancer.classpath"/>
```

Full example

This example is composed of three files, *build.xml*, *module.properties* and *module.xml*. In the file *build.xml* we load the properties for the module and call the *distribution* target defined in the *module.xml* file, which compiles, enhance and create an archive file. In the *module.properties* file we define the variables to point to the location of files.

```
########################################
# module.properties
########################################

########################################
# project
########################################
Name=my-module-name
name=${Name}
version=1.0

########################################
# ANT
########################################
build.default.target = distribution

########################################
# PROJECT
########################################
project.build.debug=on
project.lib.dir=lib

########################################
# SAMPLE FOR WAR MODULE
########################################
module.dir=./WebContent
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
module.lib.dir=${module.dir}/WEB-INF/lib
module.lib2.dir=/home/user/lib
module.classes.dir=${module.dir}/WEB-INF/classes
module.src.java.dir=./src/java
module.build.debug=on
module.archive.dist.dir=dist
module.archive.dist.file=${name}.war
module.archive.files=**/*

#######################################
# SAMPLE FOR JAR MODULE (commented out)
#######################################
#module.dir=./
#module.lib.dir=${module.dir}/lib
#module.lib2.dir=/home/user/lib
#module.classes.dir=${module.dir}/classes
#module.src.java.dir=./src
#module.build.debug=on
#module.archive.dist.dir=dist
#module.archive.dist.file=${name}.jar
#module.archive.files=**/*
```

```
<!--
===================================================================
build.xml
===================================================================
-->
<project name="project" default="default">

    <import file="module.xml"/>

    <!--
    ===================================================================
    environment
    ===================================================================
    -->
    <property environment="env"/>
    <property name="project.location" location="."/>

    <!--
    ===================================================================
    TARGET : default
    ===================================================================
    -->
    <target name="default">
        <echo
message="==================================================================="/>
        <echo message="Welcome to the build."/>
        <echo
message="==================================================================="/>
        <property file="${project.location}/module.properties"/>
        <antcall target="distribution"/>
    </target>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
</project>
```

```
<!--
===================================================================
module.xml
===================================================================
-->
<project name="module" default="distribution">

    <!--
    ===================================================================
    environment
    ===================================================================
    -->
    <property environment="env"/>
    <property name="project.location" location="."/>

    <!--
    ===================================================================
    TARGET : default
    ===================================================================
    -->
    <target name="default">
        <echo
message="===================================================================="/>
        <echo message="Welcome to the build."/>
        <echo
message="===================================================================="/>
        <antcall target="${build.default.target}"/>
    </target>

    <!--
    ===================================================================
    CONFIGURATION: MODULE
    ===================================================================
    -->

    <!-- libs necessary to build the module -->
    <path id="module.lib.classpath">
        <fileset dir="${module.lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
        <fileset dir="${module.lib2.dir}">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <!-- the classpath to compile -->
    <path id="module.compile.classpath">
        <pathelement location="${module.classes.dir}"/>
        <path refid="module.lib.classpath"/>
    </path>

    <!--
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    ==================================================================
    TARGET : clean
    ==================================================================
    -->
    <target name="clean">
        <delete includeEmptyDirs="true" quiet="true">
            <fileset dir="${module.classes.dir}"
includes="**/*.class,**/*.properties,**/*.*"/>
        </delete>
    </target>

    <!--
    ==================================================================
    TARGET : prepare
    ==================================================================
    -->
    <target name="prepare">
        <mkdir dir="${module.classes.dir}"/>
    </target>

    <!--
    ==================================================================
    TARGET : compile
    ==================================================================
    -->
    <target name="compile" depends="clean,prepare">
        <echo
message="=================================================================="/>
        <echo message="MODULE ${module.name} Compile configuration:"/>
        <echo message="java.dir       = ${module.src.java.dir}"/>
        <echo message="classes.dir    = ${module.classes.dir}"/>
        <echo
message="=================================================================="/>
        <javac srcdir="${module.src.java.dir}"
               destdir="${module.classes.dir}"
               debug="${module.build.debug}"
               classpathref="module.compile.classpath">
            <include name="**/*.java"/>
        </javac>

    </target>

    <!--
    ==================================================================
    TARGET : copy metadata files
    ==================================================================
    -->
    <target name="copy.metadata">
        <copy todir="${module.classes.dir}">
            <fileset dir="${module.src.java.dir}"
                     includes="**/*.jdo,**/*.properties,**/*.dtd,**/*.xml"/>
        </copy>
    </target>

    <!--
    ==================================================================
    TARGET : enhance
    ==================================================================
    -->
    <target name="enhance" depends="compile,copy.metadata">

        <echo
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
message="=================================================================="/>
        <echo message="Enhancing the MODULE files"/>
        <echo
message="=================================================================="/>

        <!-- the classpath to enhance -->
        <path id="module.enhancer.classpath">
            <pathelement location="${module.classes.dir}"/>
            <path refid="module.lib.classpath"/>
        </path>

        <!-- define the task enhancer -->
        <taskdef name="nucleusenhancer"
                 classpathref="module.enhancer.classpath"
                 classname="org.datanucleus.enhancer.tools.EnhancerTask" />

        <!-- enhance -->
        <nucleusenhancer classpathref="module.enhancer.classpath"/>

    </target>

    <!--
    ===================================================================
    TARGET : modulearchive
    ===================================================================
    -->
    <target name="modulearchive" depends="enhance">
        <mkdir dir="${module.archive.dist.dir}"/>
        <delete file="${module.archive.dist.dir}/${module.archive.dist.file}"/>
        <zip zipfile="${module.archive.dist.dir}/${module.archive.dist.file}">
            <zipfileset  dir="${module.classes.dir}"  prefix="" includes="**/*"/>
            <zipfileset  dir="${module.dir}"  prefix=""
includes="${module.archive.files}"/>
        </zip>
    </target>

    <!--
    ===================================================================
    TARGET : distribution
    ===================================================================
    -->
    <target name="distribution" depends="modulearchive">
        <echo message=""/>
        <echo
message="=================================================================="/>
        <echo message="Module file ready at:
${module.archive.dist.dir}/${module.archive.dist.file}"/>
        <echo
message="=================================================================="/>
        <echo message=""/>
    </target>
</project>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

19.8 User Types Extension

DataNucleus - Tutorial for Extending the DataNucleus Supported Types

Introduction

DataNucleus is an extensible persistence tool and allows the user to plug-in user extensions contributing to several persistence aspects. One of these aspects is the Java Types supported for persistence. Infinite types can be persisted with DataNucleus, and many of them are effortless. However, DataNucleus does not know all types beforehand and in situations like user defined types, mapping classes must be defined.

In this tutorial we will create a mapping class for the *java.lang.StringBuffer* (Note that this type is presented only as matter of example, since it is already supported by DataNucleus). Lets we follow the steps of defining a mapping class for a custom type.

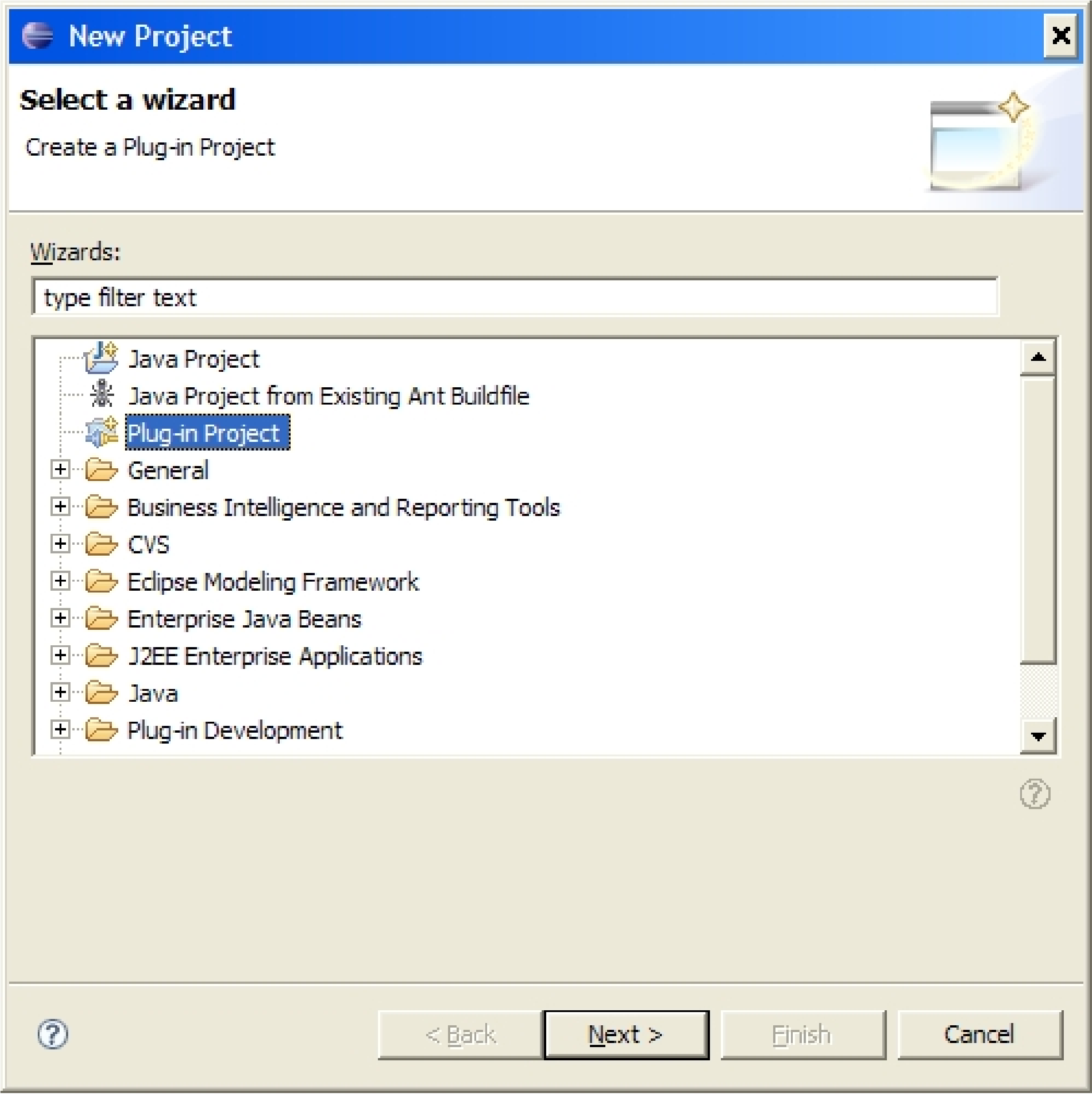
1. Step 1 : Download DataNucleus and install it as Eclipse Plug-in
2. Step 2 : Create a Plug-in Project in Eclipse IDE
3. Step 3 : Add DataNucleus Core Plug-in as dependency to the Eclipse Plug-in
4. Step 4 : Implement the *org.jpox.store_mapping* extension
5. Step 5 : Export the plug-in as a jar file
6. Step 6 : Run your application

Step 1 : Download DataNucleus and install it as Eclipse Plug-in

Download Eclipse IDE if necessary, and download DataNucleus Core jar and copy it to the Eclipse Plug-in folder *<ECLIPSE_HOME>/plugins*. Restart Eclipse when done.

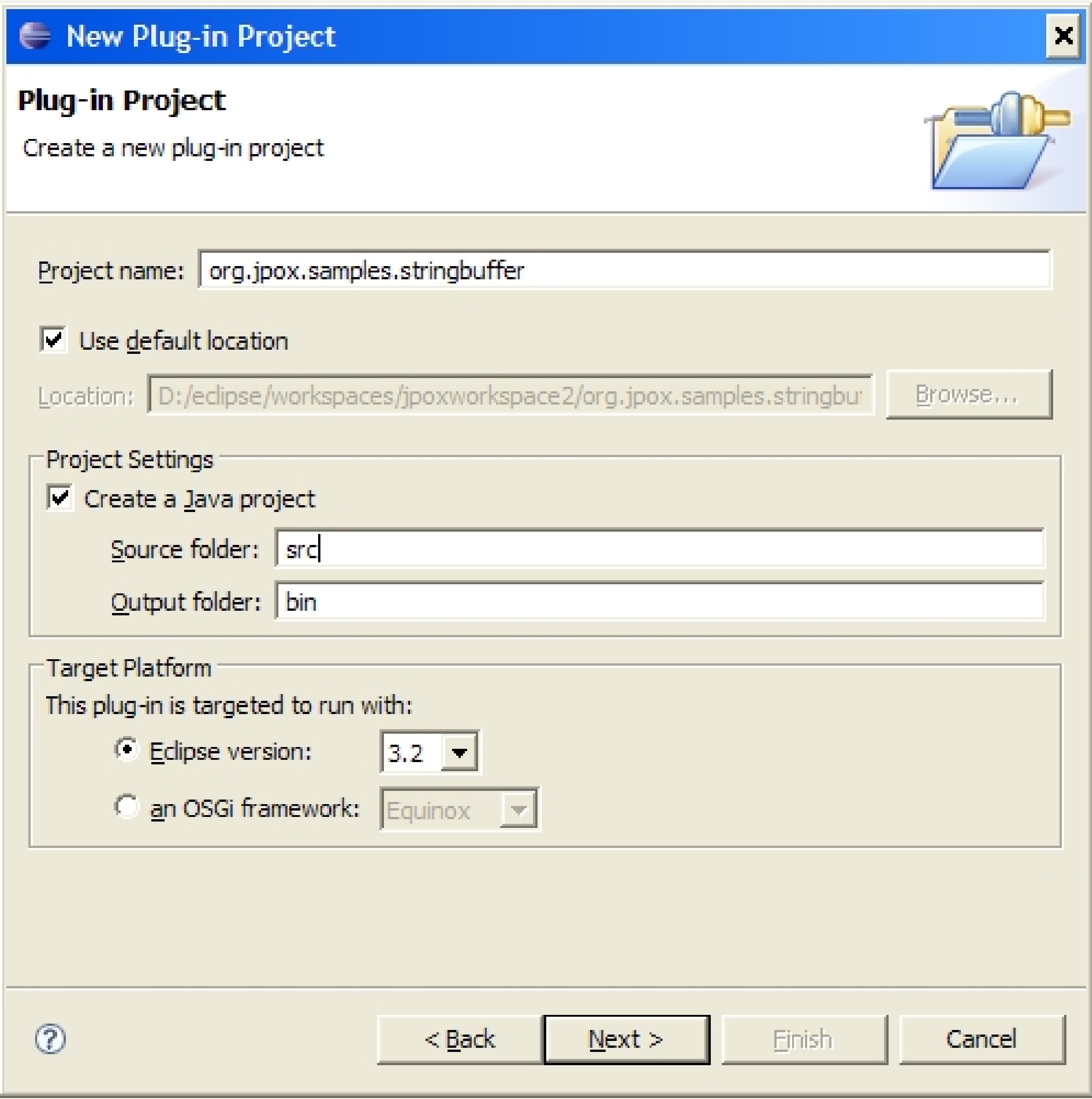
Step 2 : Create a Plug-in Project in Eclipse IDE

Create a Plug-in Project in Eclipse IDE. In Eclipse menu select *File* >> *New* >> *Project*, and follow the wizard.



Select Plug-in Project and click *Next*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Give a name to plug-in and click *Next*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

New Plug-in Project

Plug-in Content

Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID: org.jpox.samples.stringbuffer

Plug-in Version: 1.0.0

Plug-in Name: Samples StringBuffer

Plug-in Provider:

Classpath:

Plug-in Options

Generate an activator, a Java class that controls the plug-in's life cycle

Activator: sample_stringbuffer.Activator

This plug-in will make contributions to the UI

Rich Client Application

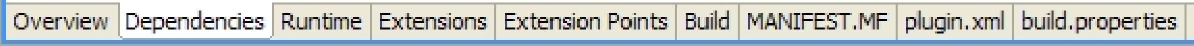
Would you like to create a rich client application? Yes No

< Back | Next > | Finish | Cancel

Unckeck the plug-in options and click on *Finish*.

Step 3 : Add DataNucleus Core Plug-in as dependency to the Eclipse Plug-in

Using the *Package Explorer* of Eclipse open the */META_INF/MANIFEST.MF* file.



Go to the *Dependencies* folder.

ALL RIGHTS RESERVED

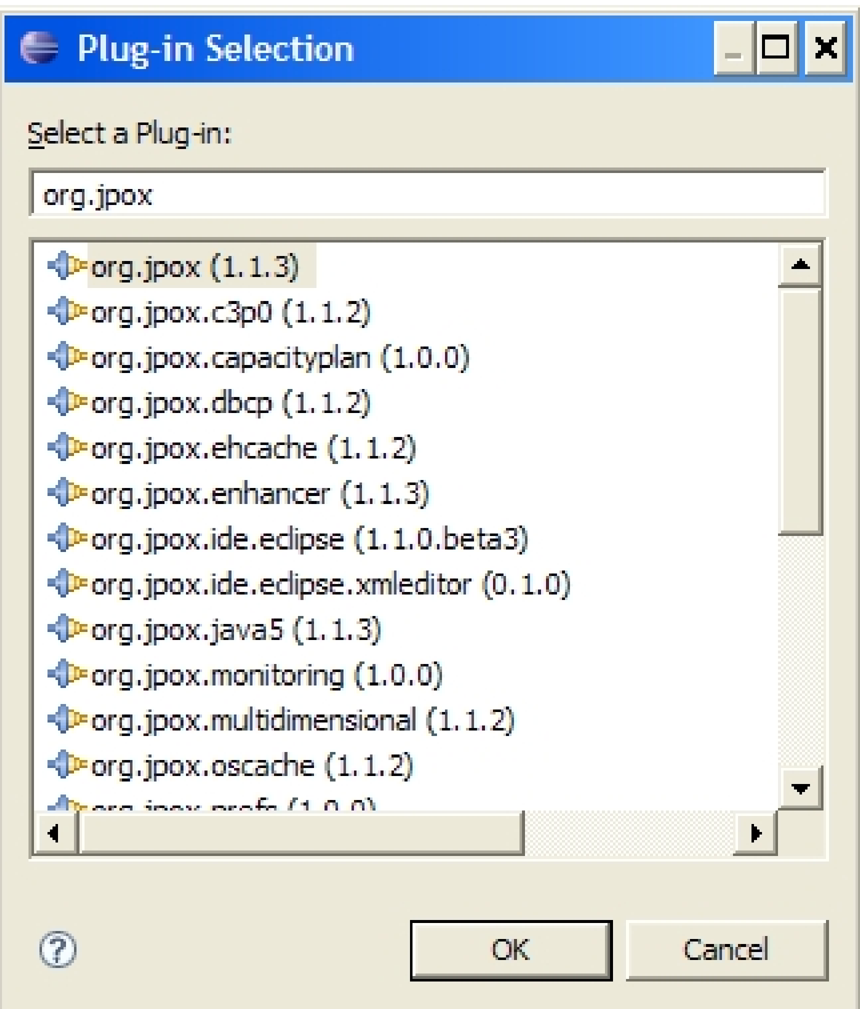
Dependencies

Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in:

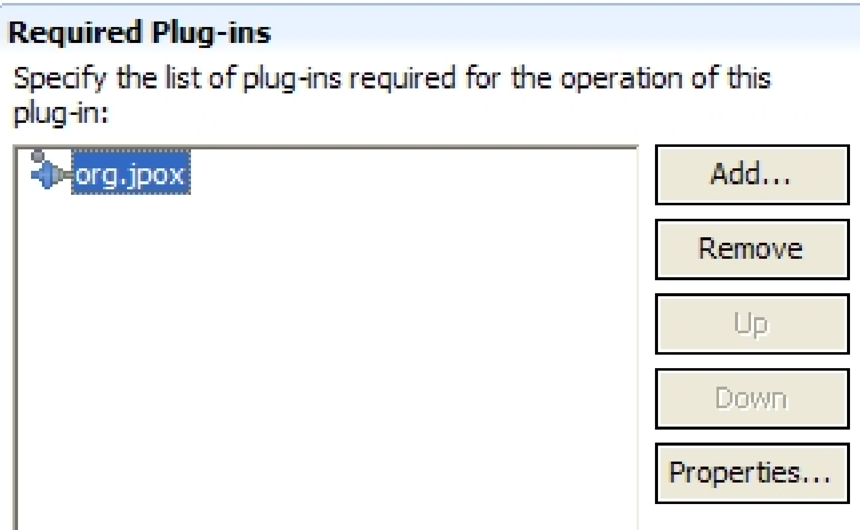
Add...
Remove
Up
Down
Properties...

Click on *Add*.



Select *org.jpox* as dependency.

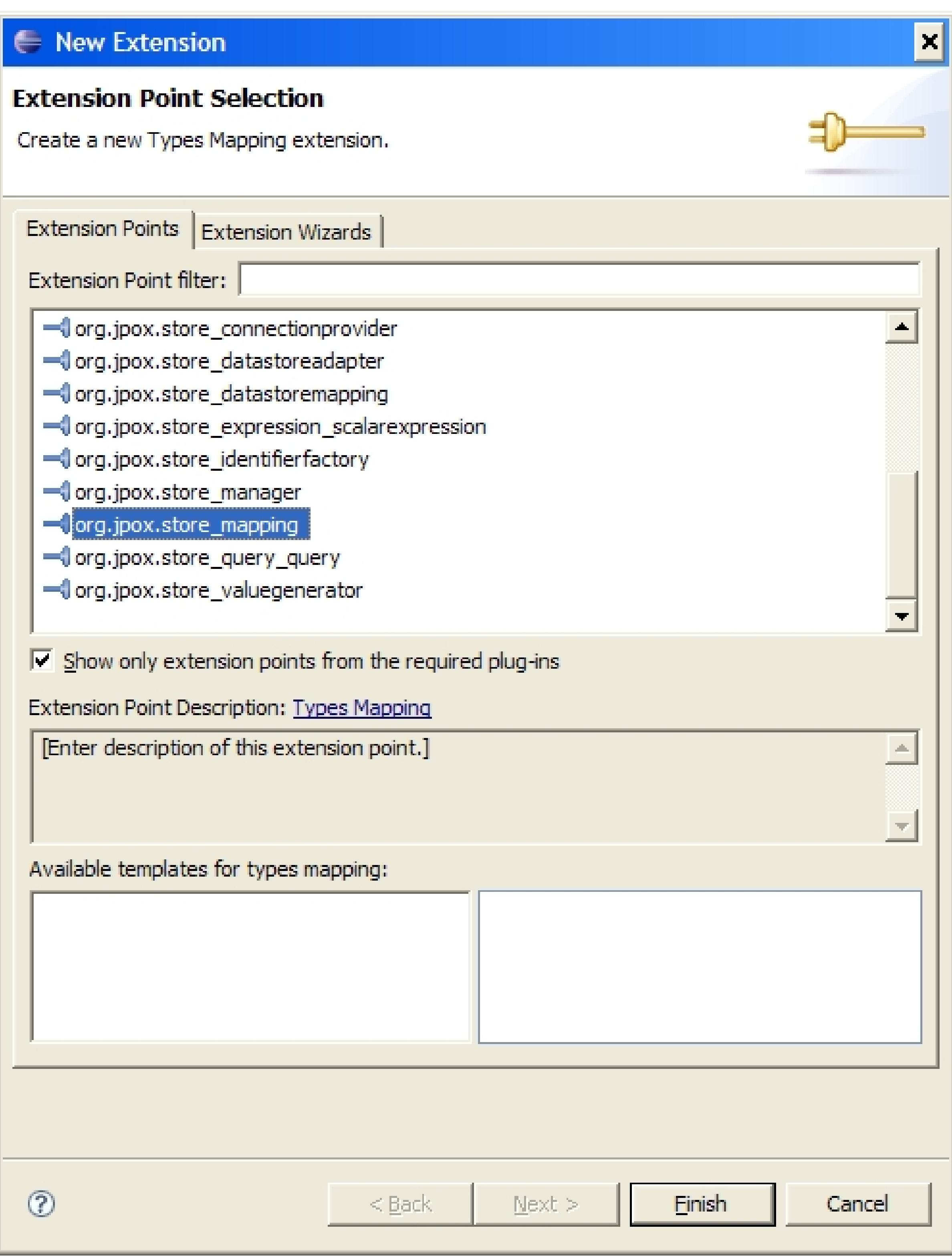
©2008-2009, DataNucleus • ALL RIGHTS RESERVED



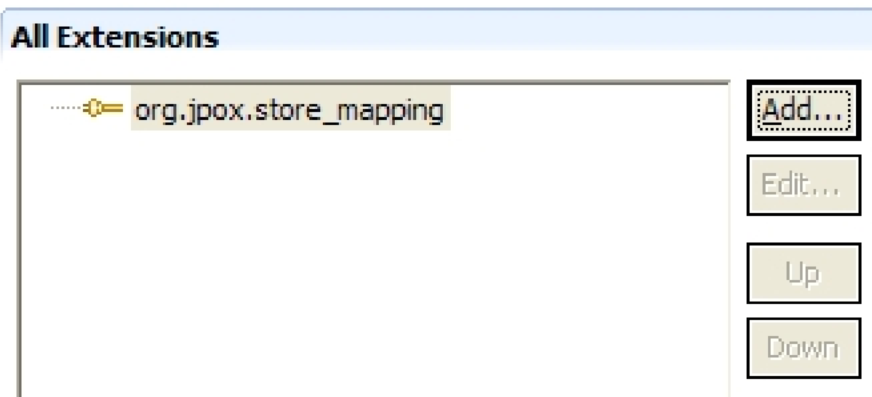
Step 4 : Implement the org.jpox.store_mapping extension

Using the *Package Explorer* of Eclipse open the */META_INF/MANIFEST.MF* file and add the DataNucleus Core Plug-in (org.jpox) as dependency to your plug-in.

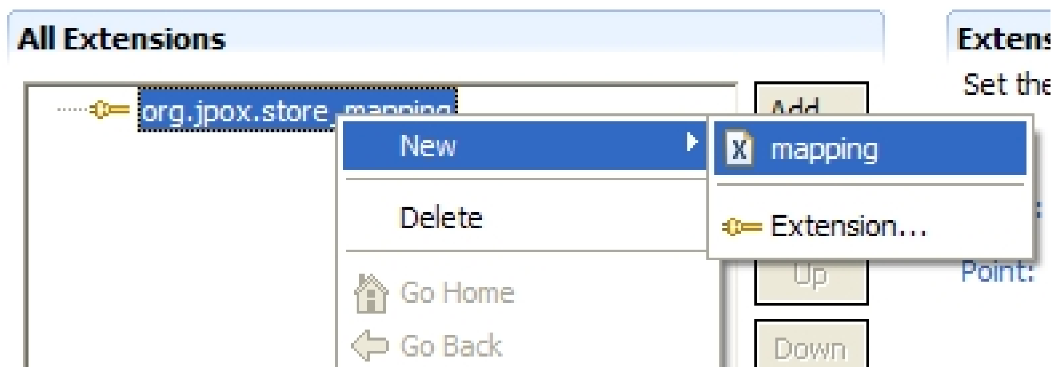
Click on *Add*.



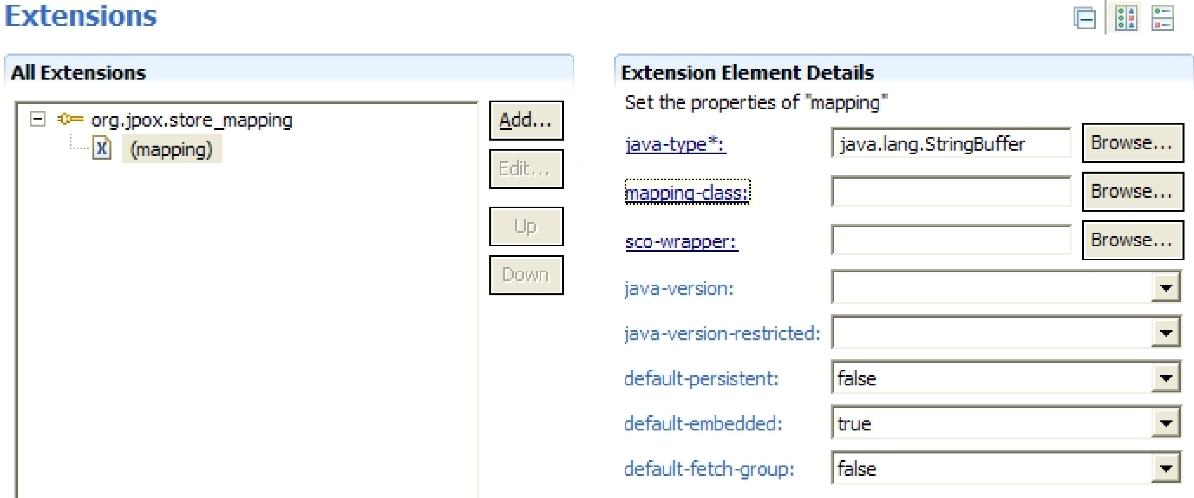
Select *org.jpox.store_mapping* and click on *Finish*.



Right click on *org.jpox.store_mapping* item.

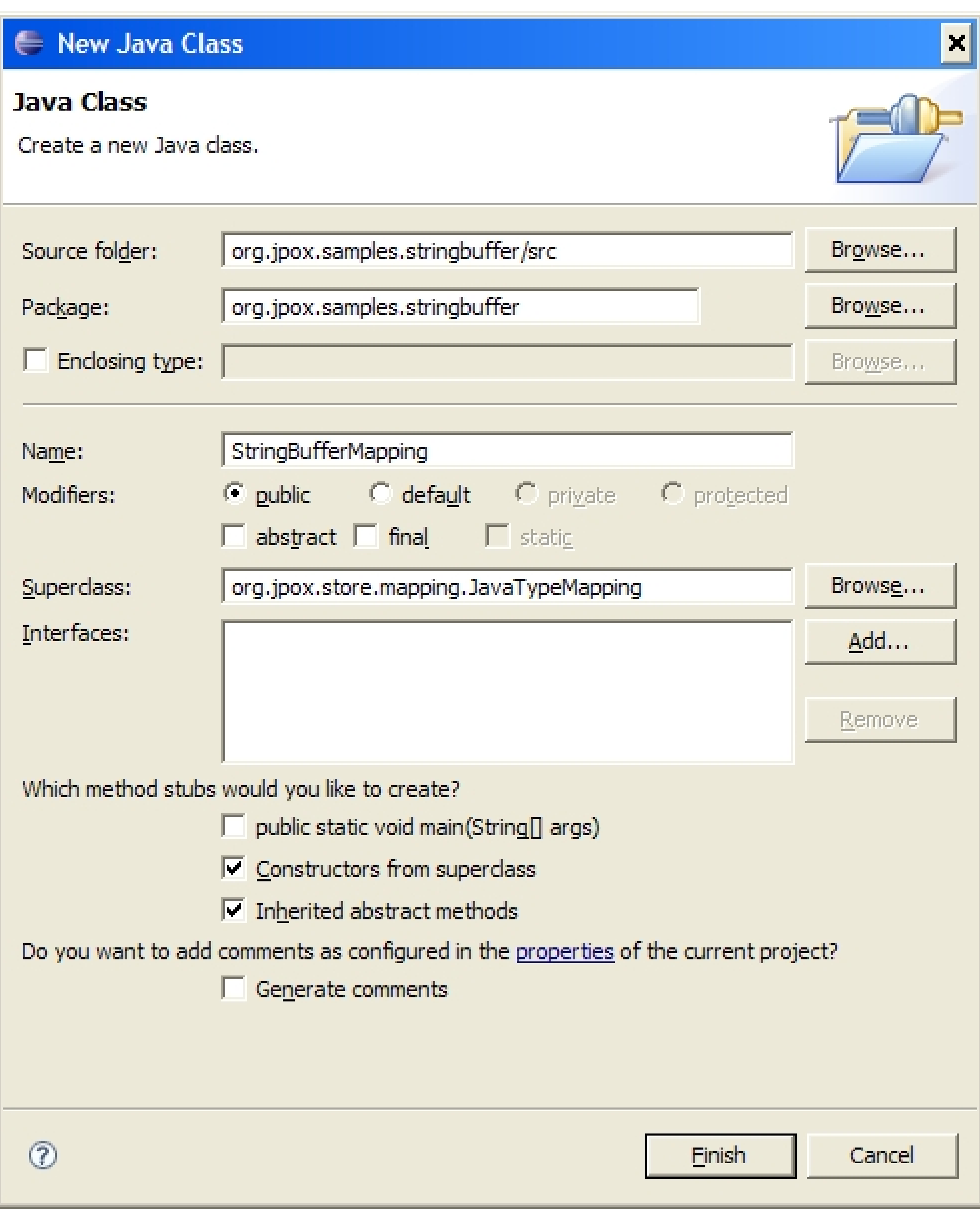


Select *New* >> *mapping*.



On the right panel, set the *java-type* field to *java.lang.StringBuffer* and click on *mapping-class* to create a new JavaTypeMapping class.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Give a class and package name and click on *Finish*.

The below code is a skeleton created automatically after this wizard, and you must implement it properly. See RDBMS User Types for additional information.

```
...
public class StringBufferMapping extends JavaTypeMapping
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
{
    public StringBufferMapping()
    {
    }

    public StringBufferMapping(DatastoreAdapter dba, String type,
AbstractPropertyMetaData fmd, DatastoreContainerObject container)
    {
        super(dba, type, fmd, container);
    }

    public Class getJavaType()
    {
        return null;
    }

    public Object getSampleValue(ClassLoaderResolver clr)
    {
        return null;
    }

    public ScalarExpression newLiteral(QueryExpression qs, Object value)
    {
        return null;
    }

    public ScalarExpression newScalarExpression(QueryExpression qs,
LogicSetExpression te)
    {
        return null;
    }
}
```

Extension Element Details

Set the properties of "mapping"

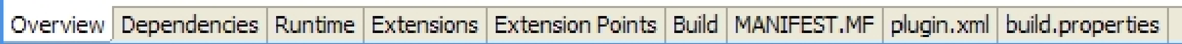
java-type*:	java.lang.StringBuffer	Browse...
mapping-class:	org.jpox.samples.stringbuffer.StringBufferMapping	Browse...
sco-wrapper:		Browse...
java-version:		
java-version-restricted:		
default-persistent:	false	
default-embedded:	true	
default-fetch-group:	false	

The results should be like the above picture. We will not create a *sco-wrapper* class because the *java.lang.StringBuffer* is a *final* class and cannot be subclassed. SCO wrapper classes are used for dirty checking SCO types.

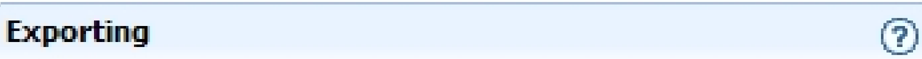
Step 5 : Export the plug-in as a jar file

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Eclipse IDE comes with an export utility that creates the jar file with the DataNucleus plug-in.



Go to the *Overview* folder.



To package and export the plug-in:

1. Organize the plug-in using the Organize Manifests Wizard
2. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
3. Export the plug-in in a format suitable for deployment using the Export Wizard

Click on *Export Wizard*.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Export

Deployable plug-ins and fragments

Export the selected projects into a form suitable for deploying in an Eclipse product

Available Plug-ins and Fragments:

- [x] org.jpox.samples.stringbuffer (1.0.0)
- [ ] org.jpox.scalability (1.0.0)
- [ ] org.jpox.shell (1.1.2)
- [ ] org.jpox.spatial (1.1.2)
- [ ] org.jpox.spatialnew (1.1.2)
- [ ] org.jpox.spatialoracle (1.1.2)
- [ ] org.jpox.swarmcache (1.1.2)
- [ ] org.jpox.tangosolcache (1.1.2)

Select All | Deselect All | Working Set...

1 out of 35 selected.

Destination | Options | JAR Signing

(•) Directory:

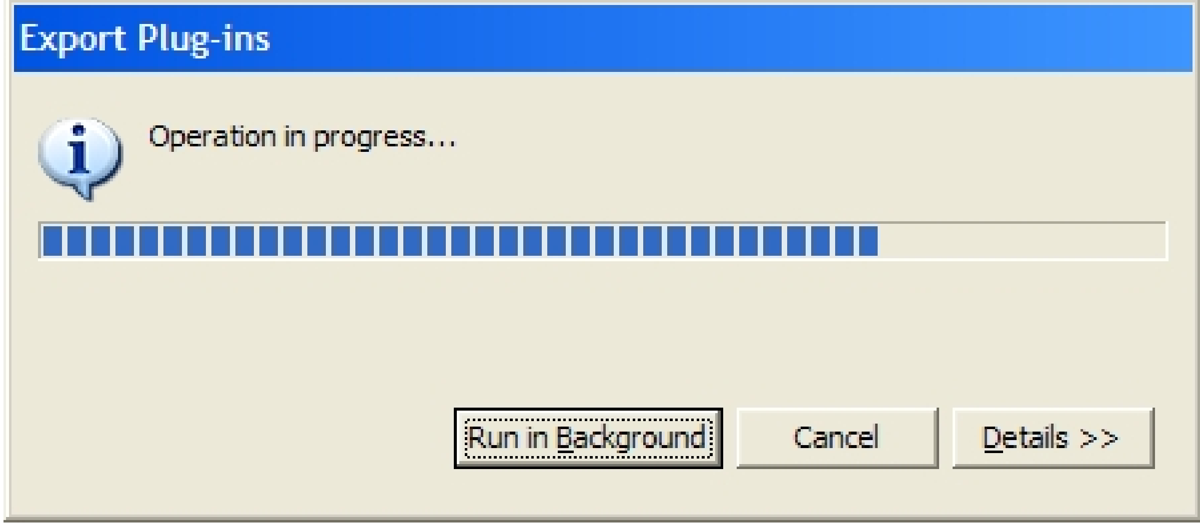
C:\Documents and Settings\ebengtso\Desktop | Browse...

( ) Archive file:

Browse...

Finish | Cancel

Select your plug-in, destination directory and click on *Finish*.



ALL RIGHTS RESERVED

Once the export process is finished the jar is created under the *<destination directory>/plugins*.

The jar contains the files:

- /META-INF/manifest.mf
- /plugin.xml
- /org/jpox/samples/stringbuffer/StringBufferMapping.class

The *manifest.mf* file looks like:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Stringbuffer Plug-in
Bundle-SymbolicName: org.jpox.samples.stringbuffer;singleton:=true
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Require-Bundle: org.jpox
```

The *plugin.xml* looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="org.jpox.store_mapping">
      <mapping
            default-embedded="true"
            default-fetch-group="false"
            default-persistent="false"
            java-type="java.lang.StringBuffer"
            mapping-class="org.jpox.samples.stringbuffer.StringBufferMapping"/>
   </extension>
</plugin>
```

Step 6 : Run your application

As any other java application you must to add the jar to the classpath before running, and the plug-in will be automatically registered with DataNucleus.

If for some reason you don't get the results expected, you can troubleshoot by enabling *DataNucleus.Plugin* logging and looking at the output.

This is the end of the tutorial.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.1 The JDO Tutorial

DataNucleus - Tutorial for JDO

Background

An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. Step 1 : Design your domain/model classes as you would do normally
2. Step 2 : Define their persistence definition using Meta-Data.
3. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
4. Step 4 : Generate the database tables where your classes are to be persisted.
5. Step 5 : Write your code to persist your objects within the DAO layer.
6. Step 6 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-tutorial-*").

Step 1 : Create your domain/model classes

Do this as you would normally. The only JDO constraint on any Java class that needs persisting is that it has a default constructor (this can be private if you prefer, and will actually be added by the DataNucleus Enhancer if you don't add it ;-)). To give a working example, let us consider an application handling products in a store.

```
package org.datanucleus.samples.jdo.tutorial;
public class Product
{
    String name = null;
    String description = null;
    double price = 0.0;

    protected Product()
    {
    }

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.jdo.tutorial;
```

```
public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
```

So we have inheritance between 2 classes. Some data in the store will be of type *Product*, and some will be *Book*. This allows us to extend our store further in the future and provide *DVD* items for example, and so on. In traditional persistence using, for example EJB CMP, this cannot be persisted directly. Instead the developer would have to generate a level above the entity beans to define the inheritance. This is messy. With JDO, we don't have this problem - we can simply throw objects across to JDO and it will persist them, and allow them to be retrieved maintaining their inheritances.

Step 2 : Define the Persistence for your classes

You now need to define how the classes should be persisted, in terms of which fields are persisted etc. This is performed by writing a Meta-Data persistence definition for each class in a JDO MetaData file. There are several ways to do this, including using XDoclet, however the most common way is to write the Meta-Data file manually. For example, for our 2 domain classes

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.datanucleus.samples.jdo.tutorial">
        <class name="Product" identity-type="datastore">
            <inheritance strategy="new-table"/>
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="price" persistence-modifier="persistent"/>
        </class>

        <class name="Book" identity-type="datastore">
            <inheritance strategy="new-table"/>
            <field name="isbn" persistence-modifier="persistent">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
            <field name="author" persistence-modifier="persistent">
                <column length="40" jdbc-type="VARCHAR"/>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            </field>
            <field name="publisher" persistence-modifier="persistent">
                <column length="40" jdbc-type="VARCHAR"/>
            </field>
        </class>
    </package>
</jdo>
```

With JDO you have various options as far as where these MetaData files are placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file *package.jdo*, in the package these classes are in.

In this tutorial we are using datastore identity which means that all objects of these classes will be assigned an identity by DataNucleus to be able to reference them. You should read about datastore identity and application identity when designing your systems persistence.

Step 3 : Instrument/Enhance your classes

JDO relies on the classes that you want to persist being PersistenceCapable. That is, they need to implement this Java interface. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable.

DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes for use by any JDO implementation. You will need to obtain the Enhancer JAR (as well as the SUN JDO JAR of course) to use this.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored

```
src/java/org/datanucleus/samples/jdo/tutorial/package.jdo
src/java/org/datanucleus/samples/jdo/tutorial/Book.java
src/java/org/datanucleus/samples/jdo/tutorial/Product.java

target/classes/org/datanucleus/samples/jdo/tutorial/package.jdo
target/classes/org/datanucleus/samples/jdo/tutorial/Book.class
target/classes/org/datanucleus/samples/jdo/tutorial/Product.class

lib/jdo2-api.jar
lib/datanucleus-core.jar
lib/datanucleus-enhancer.jar
lib/asm.jar
```

The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven project to do this for you.

```
Using Ant :
ant compile
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Using Maven :
maven java:compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

```
Using Ant :
ant enhance

Using Maven : (this is usually done automatically after the java:compile goal)
maven datanucleus:enhance


Manually on Linux/Unix :
java -cp target/classes:lib/datanucleus-enhancer.jar:lib/datanucleus-core.jar:
         lib/jdo2-api.jar:lib/asm.jar
     org.datanucleus.enhancer.DataNucleusEnhancer
     target/classes/org/datanucleus/samples/jdo/tutorial/package.jdo

Manually on Windows :
java -cp target\classes;lib\datanucleus-enhancer.jar;lib\datanucleus-core.jar;
         lib\jdo2-api.jar;lib\asm.jar
     org.datanucleus.enhancer.DataNucleusEnhancer
     target\classes\org\datanucleus\samples\jdo\tutorial\package.jdo

[Command shown on many lines to aid reading - should be on single line]
```

This command enhances the .class files that are defined by the package.jdo file. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a *ClassNotPersistenceCapableException* thrown.

You can alternatively build your application using either Maven or Ant instead of the above manual method. The use of the enhancer with these 2 build systems are documented in the Enhancer Guide

The output of this step are a set of class files that represent PersistenceCapable classes.

Step 4 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the *datanucleus.properties* file with your database details. Here we have a sample file (for MySQL)

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/myDB
javax.jdo.option.ConnectionUserName={login}
javax.jdo.option.ConnectionPassword={password}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
datanucleus.autoCreateSchema=true
datanucleus.validateTables=false
datanucleus.validateConstraints=false
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this

```
Using Ant :
ant createschema


Using Maven :
maven datanucleus:schema-create


Manually on Linux/Unix :
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-rdbms.jar:
         lib/jdo2-api.jar:lib/{jdbc_driver.jar}
     org.datanucleus.store.rdbms.SchemaTool
     -props datanucleus.properties
     -create
     target/classes/org/datanucleus/samples/jdo/tutorial/package.jdo

Manually on Windows :
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-rdbms.jar;
         lib\jdo2-api.jar;lib\{jdbc_driver.jar}
     org.datanucleus.store.rdbms.SchemaTool
     -props datanucleus.properties
     -create
     target\classes\org\datanucleus\samples\jdo\tutorial\package.jdo

[Command shown on many lines to aid reading. Should be on single line]
```

This will generate the required tables, indexes, and foreign keys for the classes defined in the JDO Meta-Data file.

Step 5 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
PersistenceManager pm = pmf.getPersistenceManager();
```

So we are creating a PersistenceManagerFactory using the file *datanucleus.properties* as used above for

DataNucleus SchemaTool. This will contain all properties necessary for our persistence usage. This file is found at the root of the CLASSPATH.

Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();

    Product product = new Product("Sony Discman","A standard discman from
Sony",49.99);
    pm.makePersistent(product);

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Please note that the *finally* step is important in that it tidies up connections to the datastore and the PersistenceManager.

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.

```
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();

    Extent e = pm.getExtent(org.datanucleus.jdo.tutorial.Product.class,true);
    Query q = pm.newQuery(e,"price < 150.00");
    q.setOrdering("price ascending");

    Collection c = (Collection)q.execute();
    Iterator iter = c.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();

        ... (use the retrieved objects)
    }

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        tx.rollback();
    }

    pm.close();
}
```

If you want to delete an object from persistence, you would perform an operation something like

```
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    ... (retrieval of objects etc)

    pm.deletePersistent(product);

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    pm.close();
}
```

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JDO book will provide many examples.

Step 6 : Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any properties file for the PersistenceManagerFactory creation
- The JDO MetaData files for your persistable classes
- Any JDBC driver classes needed for accessing your datastore
- The JDO2 API JAR (defining the JDO2 interface)
- The **DataNucleus Core** and **DataNucleus RDBMS** JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the **DataNucleus** Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

A sample *datanucleus.properties* could be like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName=org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionURL=jdbc:hsqldb:mem:nucleus
javax.jdo.option.ConnectionUserName=sa
javax.jdo.option.ConnectionPassword=

datanucleus.autoCreateSchema=true
datanucleus.validateTables=false
datanucleus.validateConstraints=false
```

```
Using Ant (you need the included "datanucleus.properties" to specify your database)
ant run


Using Maven ("runtutorial" goal included in the download JAR):
maven runtutorial


Manually on Linux/Unix :
java -cp lib/jdo2-api.jar:lib/datanucleus-core.jar:lib/datanucleus-rdbms.jar:
         lib/mysql-connector-java.jar:target/classes/:.
org.datanucleus.samples.jdo.tutorial.Main


Manually on Windows :
java -cp lib\jdo2-api.jar;lib\datanucleus-core.jar;lib\datanucleus-rdbms.jar;
         lib\mysql-connector-java.jar;target\classes\;.
org.datanucleus.samples.jdo.tutorial.Main


Output :

DataNucleus Tutorial
=============
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
>  Product : Sony Discman [A standard discman from Sony]
>  Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
>  Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial
```

Any questions?

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Again, you can download the sample classes from this tutorial from SourceForge.

The DataNucleus Team

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.2 JDO Tutorial with DB4O

DataNucleus - Tutorial for JDO using DB4O

Background

We saw in the JDO Tutorial the overall process for adding persistence to a simple application. In that tutorial we persisted the objects to an RDBMS. Here we show the differences when persisting instead to the db4o object datastore.

1. Step 1 : Design your domain/model classes as you would do normally
2. Step 2 : Define their persistence definition using Meta-Data.
3. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
4. Step 4 : Generate the database tables where your classes are to be persisted.
5. Step 5 : Write your code to persist your objects within the DAO layer.
6. Step 6 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-tutorial-*").

Step 1 : Create your domain/model classes

The model classes are exactly the same here, so please refer to the JDO Tutorial for details.

Step 2 : Define the Persistence for your classes

When defining the persistence of the classes, the only difference would be that we could omit the <column> tags from the MetaData since they are for relational datastores only and db4o would make no use of them. It would, however, be perfectly safe to leave them in and DataNucleus will ignore them. For completeness, here is the MetaData without any ORM components.

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.datanucleus.samples.jdo.tutorial">
        <class name="Product" identity-type="datastore">
            <field name="name"/>
            <field name="description"/>
            <field name="price"/>
        </class>

        <class name="Book" identity-type="datastore">
            <field name="isbn"/>
            <field name="author"/>
            <field name="publisher"/>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    </package>
</jdo>
```

Step 3 : Instrument/Enhance your classes

In the JDO Tutorial we saw that we need to bytecode enhance our classes to be persisted. This step is identical when using db4o so please refer to the original tutorial for details.

Step 4 : Generate any schema required for your domain classes

With db4o we have no such concept as a "schema" and so this step is omitted.

Step 5 : Write the code to persist objects of your classes

We saw in the JDO Tutorial how to persist our objects using JDO API calls. This is the same when using db4o so please refer to the original tutorial for details.

Step 6 : Run your application

We saw in the JDO Tutorial how to run our JDO-enabled application. This is very similar when persisting to db4o. The only differences are

- Put your db4o.jar in the CLASSPATH instead of the JDBC driver
- Put the **DataNucleus DB4O** JAR in the CLASSPATH instead of **DataNucleus RDBMS** JAR
- Edit *datanucleus.properties* to include the details for your DB4O datastore

A sample *datanucleus.properties* could be like this

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionURL=db4o:file:db4o.db
```

Any questions?

As you can see changing between an RDBMS and db4o is trivial. You don't need to change your actual classes, or persistence code at all. It's simply a change to your runtime details.

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.3 1-N Bidir FK Relation

JDO Guides : 1-N Bidirectional Relation using Foreign-Key

This guide demonstrates a 1-N collection relationship between 2 classes. In this sample we have Pack and Card such that each Pack can contain many Cards. In addition each Card has a Pack that it belongs to. We demonstrate the classes themselves, and the MetaData necessary to persist them to the datastore in the way that we require. In this case we are going to persist the relation to an RDBMS using a ForeignKey.

1. **Classes** - Design your Java classes to represent what you want to model in your system. Persistence doesn't have much of an impact on this stage, but we'll analyse the very minor influence it does have.
2. **Object Identity** - Decide how the identities of your objects of these classes will be defined. Do you want JDO to give them id's or will you do it yourself.
3. **Meta-Data** - Define how your objects of these classes will be persisted.
    1. **New Database Schema** - you have a clean sheet of paper and can have them persisted with no constraints.
    2. **Existing Database Schema** - you have existing tables that you need the objects persisted to.

The Classes

Lets look at our initial classes for the example. We want to represent a pack of cards.

```
package org.jpox.samples.packofcards.inverse;

public class Pack
{
    String name=null;
    String description=null;

    Set    cards=new HashSet();

    public Pack(String name, String desc)
    {
        this.name = name;
        this.description = desc;
    }

    public void addCard(Card card)
    {
        cards.add(card);
    }

    public void removeCard(Card card)
    {
        cards.remove(card);
    }

    public Set getCards()
    {
        return cards;
    }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    public int getNumberOfCards()
    {
        return cards.size();
    }
}

public class Card
{
    String suit=null;
    String number=null;
    Pack   pack=null;

    public Card(String suit,String number)
    {
        this.suit = suit;
        this.number = number;
    }

    public String getSuit()
    {
        return suit;
    }

    public String getNumber()
    {
        return number;
    }

    public Pack getPack()
    {
        return pack;
    }

    public void setPack(Pack pack)
    {
        this.pack = pack;
    }

    public String toString()
    {
        return "The " + number + " of " + suit;
    }
}
```

The first thing that we need to do is add a default constructor. This is a requirement of JDO. This can be private if we wish, so we add

```
public class Pack
{
    private Pack()
    {
    }

    ...
}
public class Card
{
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    private Card()
    {
    }

    ...
}
```

Object Identity

The next thing to do is decide if we want to allow DataNucleus to generate the identities of our objects, or whether we want to do it ourselves. In our case we will allow DataNucleus to create the identities for our Packs and also for our Cards.

In the case of Pack there is nothing more to code since DataNucleus will handle the identities. Similarly, in the case of Card there is nothing more to add.

MetaData for New Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition in the datastore. In this section we'll describe how to persist these objects to a new database schema where we can create new tables and don't need to write to some existing table.

Some JDO tools provide an IDE to generate Meta-Data files, but DataNucleus doesn't currently. Either way it is a good idea to become familiar with the structure of these files since they define how your classes are persisted. Lets start with the header area. You add a block like this to define that the file is JDO Meta-Data

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
```

Now let's define the persistence for our Pack class. We are going to use datastore identity here, meaning that DataNucleus will assign id's to each Pack object persisted. We define it as follows

```
    <package name="org.jpox.samples.packofcards.inverse">
        <class name="Pack" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent" mapped-by="pack">
                <collection
element-type="org.jpox.samples.packofcards.inverse.Card">
                </collection>
            </field>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        </class>
```

Here we've defined that our name field will be persisted to a VARCHAR(100) column, our description field will be persisted to a VARCHAR(255) column, and that our cards field is a Collection containing org.jpox.examples.packofcards.inverse.Card objects. In addition, it specifies that there is a pack field in the Card class (the mapped-by attribute) that gives the related pack (with the Pack being the owner of the relationship). This final information is to inform DataNucleus to link the table for this class (via a foreign key) to the table for Card class. This is what is termed a ForeignKey relationship. Please refer to the 1-N Relationships Guide for more details on this. We'll discuss JoinTable relationships in a different example.

Now lets define the persistence for our Card class. We are going to use datastore identity here, meaining that DataNucleus will assign the id's for any object of type Card. We define it as follows

```
        <class name="Card" identity-type="datastore">
            <field name="suit">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="number">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="pack">
            </field>
        </class>
    </package>
```

Here we've defined that our suit field will be persisted to a VARCHAR(10) column, our number field will be persisted to a VARCHAR(20) column.

We finally terminate the Meta-Data file with the closing tag

```
</jdo>
```

MetaData for Existing Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to an existing database schema where we already have some database tables from a previous persistence mechanism and we want to use those tables (because they have data in them). Our existing tables are shown below.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

We will take the Meta-Data that was described in the previous section (New Schema) and continue from there. To recap, here is what we arrived at

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.jpox.samples.packofcards.inverse">
        <class name="Pack" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent" mapped-by="pack">
                <collection
element-type="org.jpox.samples.packofcards.inverse.Card">
                </collection>
            </field>
        </class>
        <class name="Card" identity-type="datastore">
            <field name="suit">
                <column length="10" jdbc-type="VARCHAR"/>
            </field>
            <field name="number">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
            <field name="pack">
            </field>
        </class>
    </package>
</jdo>
```

The first thing we need to do is map the Pack class to the table that we have in our database. It needs to be mapped to a table called "DECK", with columns "IDENTIFIERNAME" and "DETAILS", and the identity column that DataNucleus uses needs to be called IDENTIFIER_ID. We do this by changing the Meta-Data to be

```
        <class name="Pack" identity-type="datastore" table="DECK">
            <datastore-identity>
                <column name="IDENTIFIER_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="IDENTIFIERNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DETAILS" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent" mapped-by="pack">
                <collection
element-type="org.jpox.samples.packofcards.inverse.Card">
                </collection>
            </field>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So we made use of the attribute table (of element class) and name (of element column) to align to the table that is there. In addition we made use of the datastore-identity element to map the identity column name. Lets now dothe same for the class Card. In our database we want this to map to a table called "PLAYINGCARD", with columns "SET" and "VALUE". So we do the same thing to its Meta-Data

```
<class name="Card" identity-type="datastore" table="PLAYINGCARD">
    <datastore-identity>
        <column name="PLAYINGCARD_ID"/>
    </datastore-identity>
    <field name="suit">
        <column name="SET" length="10" jdbc-type="VARCHAR"/>
    </field>
    <field name="number">
        <column name="VALUE" length="20" jdbc-type="VARCHAR"/>
    </field>
    <field name="pack">
        <column name="DECK_ID"/>
    </field>
</class>
```

OK, so we've now mapped our 2 classes to their tables. This completes our job. The only other aspect that is likely to be met is where a column in the database is of a particular type, but we'll cover that in a different example.

One thing worth mentioning is the difference if our Collection class was a List, ArrayList, Vector, etc. In this case we need to specify the ordering column for maintaining the order within the List. In our case we want to specify this column to be called "IDX", so we do it like this.

```
<class name="Card" identity-type="datastore" table="PLAYINGCARD">
    <datastore-identity>
        <column name="PLAYINGCARD_ID"/>
    </datastore-identity>
    <field name="suit">
        <column name="SET" length="10" jdbc-type="VARCHAR"/>
    </field>
    <field name="number">
        <column name="VALUE" length="20" jdbc-type="VARCHAR"/>
    </field>
    <field name="pack">
        <column name="DECK_ID"/>
        <order column="IDX"/>
    </field>
</class>
```

Sample Code

The full code is available for download from SourceForge. If you have any queries about this example, please raise them on our Forum.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.4 1-N Bidir JoinTable Relation

JDO Guides : 1-N Bidirectional Relation using Join Table

This guide demonstrates a 1-N collection relationship between 2 classes. In this sample we have Pack and Card such that each Pack can contain many Cards. In addition each Card has a Pack that it belongs to. We demonstrate the classes themselves, and the MetaData necessary to persist them to the datastore in the way that we require. In this case we are going to persist the relation to an RDBMS using a JoinTable.

1. **Classes** - Design your Java classes to represent what you want to model in your system. Persistence doesn't have much of an impact on this stage, but we'll analyse the very minor influence it does have.
2. **Object Identity** - Decide how the identities of your objects of these classes will be defined. Do you want JDO to give them id's or will you do it yourself.
3. **Meta-Data** - Define how your objects of these classes will be persisted.
    1. **New Database Schema** - you have a clean sheet of paper and can have them persisted with no constraints.
    2. **Existing Database Schema** - you have existing tables that you need the objects persisted to.

The Classes

Let's look at our initial classes for the example. We want to represent a pack of cards.

```
package org.jpox.samples.packofcards.normal;

public class Pack
{
    String name=null;
    String description=null;

    Set    cards=new HashSet();

    public Pack(String name, String desc)
    {
        this.name = name;
        this.description = desc;
    }

    public void addCard(Card card)
    {
        cards.add(card);
    }

    public void removeCard(Card card)
    {
        cards.remove(card);
    }

    public Set getCards()
    {
        return cards;
    }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    public int getNumberOfCards()
    {
        return cards.size();
    }
}

public class Card
{
    String suit=null;
    String number=null;

    public Card(String suit,String number)
    {
        this.suit = suit;
        this.number = number;
    }

    public String getSuit()
    {
        return suit;
    }

    public String getNumber()
    {
        return number;
    }

    public String toString()
    {
        return "The " + number + " of " + suit;
    }
}
```

The first thing that we need to do is add a default constructor. This is a requirement of JDO. This can be private if we wish, so we add

```
public class Pack
{
    private Pack()
    {
    }

    ...
}
public class Card
{
    private Card()
    {
    }

    ...
}
```

Object Identity

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The next thing to do is decide if we want to allow DataNucleus to generate the identities of our objects, or whether we want to do it ourselves. In our case we will allow DataNucleus to create the identities for our Packs, but since we know which Cards are in a pack we want to give them an identity based on the suit and the number. This allows us to give examples of both datastore identity and application identity.

In the case of Pack there is nothing more to code since DataNucleus will handle the identities. With Card however we now need to define a definition of the primary key. With JDO we must define a primary key class. In our case we want our primary key to be a composite of suit and number, so we define a primary key as follows

```
public static class CardKey
{
    public String suit;
    public String number;

    /**
     *  Default constructor (mandatory for JDO).
     */
    public CardKey()
    {
    }

    /**
     * String constructor (mandatory for JDO).
     **/
    public CardKey(String str)
    {
        StringTokenizer toke = new StringTokenizer (str, "::");
        str = toke.nextToken ();
        this.suit = str;
        str = toke.nextToken ();
        this.number = str;
    }

    /**
     * Implementation of equals method (JDO requirement).
     **/
    public boolean equals(Object obj)
    {
        if (obj == this)
        {
            return true;
        }
        if (!(obj instanceof CardKey))
        {
            return false;
        }
        CardKey c=(CardKey)obj;

        return suit.equals(c.suit) &amp;&amp; number.equals(c.number);
    }

    /**
     * Implementation of hashCode (JDO requirement)
     */
    public int hashCode ()
    {
        return this.suit.hashCode() ^ this.number.hashCode();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }

        /**
         * Implementation of toString that outputs this object id's PK values.
         * (JDO requirement).
         **/
        public String toString ()
        {
            return this.suit + "::" + this.number;
        }
    }
```

We decide that we don't want to use this class as an external object so we make it an inner class of our Card class. This is optional. You could have it as a class used elsewhere if you wish.

MetaData for New Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to a new database schema where we can create new tables and don't need to write to some existing table.

Some JDO tools provide an IDE to generate Meta-Data files, but DataNucleus doesn't currently. Either way it is a good idea to become familiar with the structure of these files since they define how your classes are persisted. Lets start with the header area. You add a block like this to define that the file is JDO Meta-Data

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
```

Now lets define the persistence for our Pack class. We are going to use datastore identity here, meaning that DataNucleus will assign id's to each Pack object persisted and store them in a surrogate column in the datastore. We define it as follows

```
    <package name="org.jpox.samples.packofcards.normal">
        <class name="Pack" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent">
                <collection
element-type="org.jpox.samples.packofcards.normal.Card"/>
                <join/>
            </field>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Here we've defined that our name field will be persisted to a VARCHAR(100) column, our description field will be persisted to a VARCHAR(255) column, and that our cards field is a Collection containing org.jpox.samples.packofcards.normal.Card objects. This final information is to inform DataNucleus to link the table for this class (via a foreign key) to the table for Card class. This is what is termed a JoinTable relationship. Please refer to the 1-N Relationships Guide for more details on this. We'll discuss ForeignKey relationships in a different example.

Now lets define the persistence for our Card class. We are going to use application identity here, meaning that we store the id's for any object of type Card in a nominated field/fields, and that we can set the value if we so wish. We define it as follows

```
        <class name="Card" identity-type="application"
           objectid-class="org.jpox.samples.packofcards.normal.Card$CardKey">
            <field name="suit" primary-key="true">
                <column length="10" jdbc-type="VARCHAR"/>
            </field>
            <field name="number" primary-key="true">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
        </class>
    </package>
```

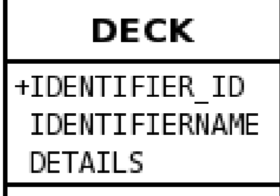
Here we've defined that our class uses application identity and that its primary key for this is org.jpox.samples.packofcards.normal.Card$CardKey (since its an inner class, we use the $ notation to define that). We've also defined that the 2 fields in this class are part of the primary key. You must define which fields are in the primary key.

We finally terminate the Meta-Data file with the closing tag

```
</jdo>
```

MetaData for Existing Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to an existing database schema where we already have some database tables from a previous persistence mechanism and we want to use those tables (because they have data in them). Our existing tables are shown below.



We will take the Meta-Data that was described in the previous section (New Schema) and continue from there. To recap, here is what we arrived at

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.jpox.samples.packofcards.normal">
        <class name="Pack" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent">
                <collection
element-type="org.jpox.samples.packofcards.normal.Card"/>
                <join/>
            </field>
        </class>
        <class name="Card" identity-type="application"
           objectid-class="org.jpox.samples.packofcards.normal.Card$CardKey">
            <field name="suit" primary-key="true">
                <column length="10" jdbc-type="VARCHAR"/>
            </field>
            <field name="number" primary-key="true">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
        </class>
    </package>
</jdo>
```

The first thing we need to do is map the Pack class to the table that we have in our database. It needs to be mapped to a table called "DECK", with columns "IDENTIFIERNAME" and "DETAILS", and the identity column that DataNucleus uses needs to be called IDENTIFIER_ID. We do this by changing the Meta-Data to be

```
        <class name="Pack" identity-type="datastore" table="DECK">
            <datastore-identity>
                <column name="IDENTIFIER_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="IDENTIFIERNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DETAILS" length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent">
                <collection
element-type="org.jpox.samples.packofcards.normal.Card"/>
                <join/>
            </field>
        </class>
```

So we made use of the attributes table (of element class) and name (of element column) to align to the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

table that is there. In addition we made use of the datastore-identity element to map the identity column name. Lets now do the same for the class Card. In our database we want this to map to a table called "PLAYINGCARD", with columns "SET" and "VALUE". So we do the same thing to its Meta-Data

```
        <class name="Card" identity-type="application" table="PLAYINGCARD"
           objectid-class="org.jpox.samples.packofcards.normal.Card$CardKey">
            <field name="suit" primary-key="true">
                <column name="SET" length="10" jdbc-type="VARCHAR"/>
            </field>
            <field name="number" primary-key="true">
                <column name="VALUE" length="20" jdbc-type="VARCHAR"/>
            </field>
        </class>
```

OK, so we've now mapped our 2 classes to their tables. The only remaining thing is that to form our relationship with the "JoinTable Relationship" we have a join table and in our database this is called "DECKOFCARDS" with columns "SET_ID", "VALUE_ID" and "DECK_ID". To map to these we need to further modify the Meta-Data for Pack as follows

```
        <class name="Pack" identity-type="datastore" table="DECK">
            <datastore-identity>
                    <column name="IDENTIFIER_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="IDENTIFIERNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DETAILS" length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent"
table="DECKOFCARDS">
                <collection
element-type="org.jpox.samples.packofcards.normal.Card"/>
                <join>
                    <column name="DECK_ID"/>
                </join>
                <element>
                    <column name="SET_ID" target="SET"/>
                    <column name="VALUE_ID" target="VALUE"/>
                </element>
            </field>
        </class>
```

So we've now updated the cards field to use the new JDO 2 attributes field:table, field:join:column, and field:element:column to have the mapping to the existing table. Please note the use of multiple columns where we have to map to the composite primary key on Card. This completes our job. The only other aspect that is likely to be met is where a column in the database is of a particular type, but we'll cover that in a different example.

One thing worth mentioning is the difference if our Collection class was a List, ArrayList, Vector, etc. In this case we need to specify the ordering column for maintaining the order within the List. In our case we want to specify this column to be called "IDX", so we do it like this.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <class name="Pack" identity-type="datastore" table="DECK">
            <datastore-identity>
                    <column name="IDENTIFIER_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="IDENTIFIERNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DETAILS" length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="cards" persistence-modifier="persistent"
table="DECKOFCARDS">
                <collection
element-type="org.jpox.samples.packofcards.normal.Card"/>
                <join>
                    <column name="DECK_ID"/>
                </join>
                <element>
                    <column name="SET_ID" target="SET"/>
                    <column name="VALUE_ID" target="VALUE"/>
                </element>
                <order column="IDX"/>
            </field>
        </class>
```

Example Code

The full code is available for download from SourceForge. If you have any queries about this example, please raise them on our Forum.

20.5 M-N Relation

JDO Guides : M-N Relation

This guide demonstrates an M-N collection relationship between 2 classes. In this sample we have Supplier and Customer such that each Customer can contain many Suppliers. In addition each Supplier can have many Customers. We demonstrate the classes themselves, and the MetaData necessary to persist them to the datastore in the way that we require.

1. **Classes** - Design your Java classes to represent what you want to model in your system. JDO doesn't have much of an impact on this, but we'll analyse the very minor influence it does have.
2. **Meta-Data** - Define how your objects of these classes will be persisted.
    1. **New Database Schema** - you have a clean sheet of paper and can have them persisted with no constraints.
    2. **Existing Database Schema** - you have existing tables that you need the objects persisted to.
3. **Managing the Relationship** - How we add/remove elements to/from the M-N relation.

The Classes

Lets look at our initial classes for the example. We want to represent the relation between a customer and a supplier.

```
package org.jpox.samples.m_to_n;

public class Customer
{
    String name = null;
    String description = null;
    Collection suppliers = new HashSet();

    public Customer(String name, String desc)
    {
        this.name = name;
        this.description = desc;
    }

    public void addSupplier(Supplier supplier)
    {
        suppliers.add(supplier);
    }

    public void removeSupplier(Supplier supplier)
    {
        suppliers.remove(supplier);
    }

    public Collection getSuppliers()
    {
        return suppliers;
    }

    public int getNumberOfSuppliers()
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    {
        return suppliers.size();
    }
}
```

```
public class Supplier
{
    String name = null;
    String address = null;
    Collection customers = new HashSet();

    public Supplier(String name, String address)
    {
        this.name = name;
        this.address = address;
    }

    public String getName()
    {
        return name;
    }

    public String getAddress()
    {
        return address;
    }

    public void addCustomer(Customer customer)
    {
        customers.add(customer);
    }

    public void removeCustomer(Customer customer)
    {
        customers.remove(customer);
    }

    public Collection getCustomers()
    {
        return customerers;
    }

    public int getNumberOfCustomers()
    {
        return customers.size();
    }
}
```

The first thing that we need to do is add a default constructor. This is a requirement of JDO. In our case we are using the DataNucleus enhancer and this will automatically add the default constructor when not present, so we omit this.

In this example we don't care about the "identity" type chosen so we will use datastore-identity. Please refer to the documentation for examples of application and datastore identity for how to specify them.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

MetaData for New Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to a new database schema where we can create new tables and don't need to write to some existing table.

Some JDO tools provide an IDE to generate Meta-Data files, but DataNucleus doesn't currently. Either way it is a good idea to become familiar with the structure of these files since they define how your classes are persisted. Lets start with the header area. You add a block like this to define that the file is JDO Meta-Data

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
```

Now let's define the persistence for our Customer class. We define it as follows

```
    <package name="org.jpox.samples.m_to_n">
        <class name="Customer" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="suppliers" persistence-modifier="persistent"
mapped-by="customers">
                <collection element-type="org.jpox.samples.m_to_n.Supplier"/>
                <join/>
            </field>
        </class>
```

Here we've defined that our name field will be persisted to a VARCHAR(100) column, our description field will be persisted to a VARCHAR(255) column, and that our suppliers field is a Collection containing org.jpox.examples.m_to_n.Supplier objects. In addition, it specifies that there is a customers field in the Supplier class (the mapped-by attribute) that gives the related customers for the Supplier. This final information is to inform DataNucleus to link the table for this class to the table for Supplier class. This is what is termed an M-N relationship. Please refer to the M-N Relationships Guide for more details on this.

Now lets define the persistence for our Supplier class. We define it as follows

```
        <class name="Supplier" identity-type="datastore">
            <field name="name">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="address">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="customers">
            <collection element-type="org.jpox.samples.m_to_n.Supplier"/>
            <join/>
        </field>
    </class>
</package>
```

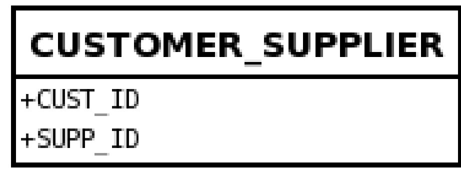
Here we've defined that our name field will be persisted to a VARCHAR(100) column, our address field will be persisted to a VARCHAR(100) column.

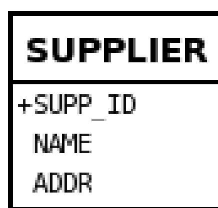
We finally terminate the Meta-Data file with the closing tag

```
</jdo>
```

MetaData for Existing Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to an existing database schema where we already have some database tables from a previous persistence mechanism and we want to use those tables (because they have data in them). Our existing tables are shown below.





We will take the Meta-Data that was described in the previous section (New Schema) and continue from there. To recap, here is what we arrived at

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.jpox.samples.m_to_n">
        <class name="Customer" identity-type="datastore">
            <field name="name" persistence-modifier="persistent">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="suppliers" persistence-modifier="persistent"
mapped-by="customers">
                <collection element-type="org.jpox.samples.m_to_n.Supplier"/>
                <join/>
            </field>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <class name="Supplier" identity-type="datastore">
            <field name="name">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="address">
                <column length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="customers">
                <collection element-type="org.jpox.samples.m_to_n.Customer"/>
                <join/>
            </field>
        </class>
    </package>
</jdo>
```

The first thing we need to do is map the Customer class to the table that we have in our database. It needs to be mapped to a table called "CUSTOMER", with columns "NAME" and "DESC", and the identity column that DataNucleus uses needs to be called CUST_ID. We do this by changing the Meta-Data to be

```
        <class name="Customer" identity-type="datastore" table="CUSTOMER">
            <datastore-identity>
                <column name="CUST_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="NAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DESC" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="suppliers" persistence-modifier="persistent"
mapped-by="customers">
                <collection element-type="org.jpox.samples.m_to_n.Supplier"/>
                <join/>
            </field>
        </class>
```

We now need to define the mapping for the join table storing the relationship information. So we make use of the "table" attribute of field to define this, and use the join and element subelements to define the columns of the join table. Like this

```
        <class name="Customer" identity-type="datastore" table="CUSTOMER">
            <datastore-identity>
                <column name="CUST_ID"/>
            </datastore-identity>
            <field name="name" persistence-modifier="persistent">
                <column name="NAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="description" persistence-modifier="persistent">
                <column name="DESC" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="suppliers" persistence-modifier="persistent"
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
mapped-by="customers" table="CUSTOMER_SUPPLIER">
                <collection element-type="org.jpox.samples.m_to_n.Supplier"/>
                <join>
                    <column name="CUST_ID"/>
                </join>
                <element>
                    <column name="SUPP_ID"/>
                </element>
            </field>
        </class>
```

Lets now dothe same for the class Supplier. In our database we want this to map to a table called "SUPPLIER", with columns "SUPP_ID" (identity), "NAME" and "ADDR". We need to do nothing more for the join table since it is shared and we have defined its table/columns above.

```
         <class name="Supplier" identity-type="datastore" table="SUPPLIER">
             <datastore-identity>
                <column name="SUPP_ID"/>
            </datastore-identity>
            <field name="name">
                <column name="NAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="address">
                <column name="ADDR" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="customers">
                <collection element-type="org.jpox.samples.m_to_n.Customer"/>
                <join/>
            </field>
        </class>
```

OK, so we've now mapped our 2 classes to their tables. This completes our job. The only other aspect that is likely to be met is where a column in the database is of a particular type, but we'll cover that in a different example.

Management of the Relation

We now have our classes and the definition of persistence and we need to use our classes in the application. This section defines how we maintain the relation between the objects. Let's start by creating a few objects

```
    PersistenceManager pm = pmf.getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    Object[] custIds = new Object[2];
    Object[] suppIds = new Object[3];
    try
    {
        tx.begin();

        Customer cust1 = new Customer("DFG Stores", "Small shop in London");
        Customer cust2 = new Customer("Kevins Cards", "Gift shop");
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        Supplier supp1 = new Supplier("Stationery Direct", "123 The boulevard, 
Milton Keynes, UK");
        Supplier supp2 = new Supplier("Grocery Wholesale", "56 Jones Industrial 
Estate, London, UK");
        Supplier supp3 = new Supplier("Makro", "1 Parkville, Wembley, UK");

        pm.makePersistent(cust1);
        pm.makePersistent(cust2);
        pm.makePersistent(supp1);
        pm.makePersistent(supp2);
        pm.makePersistent(supp3);
        tx.commit();
        custIds[0] = JDOHelper.getObjectId(cust1);
        custIds[1] = JDOHelper.getObjectId(cust2);
        suppIds[0] = JDOHelper.getObjectId(supp1);
        suppIds[1] = JDOHelper.getObjectId(supp2);
        suppIds[2] = JDOHelper.getObjectId(supp3);
    }
    catch (Exception e)
    {
        // Handle any errors
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
```

OK. We've now persisted some Customers and Suppliers into our datastore. We now need to establish the relations.

```
    PersistenceManager pm = pmf.getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    try
    {
        tx.begin();

        Customer cust1 = (Customer)pm.getObjectById(custIds[0]);
        Customer cust2 = (Customer)pm.getObjectById(custIds[1]);
        Supplier supp1 = (Supplier)pm.getObjectById(suppIds[0]);
        Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);

        // Establish the relation customer1 uses supplier2
        cust1.addSupplier(supp2);
        supp2.addCustomer(cust1);

        // Establish the relation customer2 uses supplier1
        cust2.addSupplier(supp1);
        supp1.addCustomer(cust2);

        tx.commit();
    }
    catch (Exception e)
    {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        // Handle any errors
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
```

You note that we set both sides of the relation. This is important since JDO doesnt define support for "managed relations" before JDO2.1. We could have adapted the Customer method addSupplier to add both sides of the relation (or alternatively via Supplier method addCustomer) to simplify this process.

Let's now assume that over time we want to change our relationships.

```

    PersistenceManager pm = pmf.getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    try
    {
        tx.begin();

        // Retrieve the objects
        Customer cust1 = (Customer)pm.getObjectById(custIds[0]);
        Customer cust2 = (Customer)pm.getObjectById(custIds[1]);
        Supplier supp1 = (Supplier)pm.getObjectById(suppIds[0]);
        Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);
        Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);

        // Remove the relation from customer1 to supplier2, and add relation to
supplier3
        cust1.removeSupplier(supp2);
        supp2.removeCustomer(cust1);
        cust1.addSupplier(supp3);
        supp3.addCustomer(cust1);

        // Add a relation customer2 uses supplier3
        cust2.addSupplier(supp3);
        supp3.addCustomer(cust2);

        tx.commit();
    }
    catch (Exception e)
    {
        // Handle any errors
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So now we have customer1 with a relation to supplier3, and we have customer2 with relations to supplier1 and suppier3. That should give enough idea of how to manage the relations. The most important thing with any bidirectional relation is to set both sides of the relation.

Sample Code

The full code is available for download from SourceForge. If you have any queries about this example, please raise them on our Forum.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.6 M-N Attributed Relation

JDO Guides : M-N Attributed Relation

DataNucleus provides support for standard JDO M-N relations where we have a relation between, for example, *Customer* and *Supplier*, where a *Customer* has many *Supplier*s and a *Supplier* has many *Customer*s. A slight modification on this is where you have the relation carrying some additional attributes of the relation. Let's take some classes

```
public class Customer
{
    private long id; // PK
    private String name;
    private Set supplierRelations = new HashSet();

    ...
}

public class Supplier
{
    private long id; // PK
    private String name;
    private Set customerRelations = new HashSet();

    ...
}
```

Now we obviously cant define an "attributed relation" using Java and just these classes so we invent an intermediate "associative" class, that will also contain the attributes.

```
public class BusinessRelation
{
    private Customer customer; // PK
    private Supplier supplier; // PK
    private String relationLevel;
    private String meetingLocation;

    public BusinessRelation(Customer cust, Supplier supp, String level, String
meeting)
    {
        this.customer = cust;
        this.supplier = supp;
        this.relationLevel = level;
        this.meetingLocation = meeting;
    }
    ...
}
```

So we define the metadata like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
<jdo>
    <package name="org.jpox.test">
        <class name="Customer" detachable="true" table="CUSTOMER">
            <field name="id" primary-key="true" value-strategy="increment"
column="ID"/>
            <field name="name" column="NAME"/>
            <field name="supplierRelations" persistence-modifier="persistent"
mapped-by="customer">
                <collection element-type="BusinessRelation"/>
            </field>
        </class>

        <class name="Supplier" detachable="true" table="SUPPLIER">
            <field name="id" primary-key="true" value-strategy="increment"
column="ID"/>
            <field name="name" column="NAME"/>
            <field name="customerRelations" persistence-modifier="persistent"
mapped-by="supplier">
                <collection element-type="BusinessRelation"/>
            </field>
        </class>

        <class name="BusinessRelation" type="application" detachable="true"
objectid-class="BusinessRelation$PK" table="BUSINESSRELATION">
            <field name="customer" primary-key="true" column="CUSTOMER_ID"/>
            <field name="supplier" primary-key="true" column="SUPPLIER_ID"/>
            <field name="relationLevel" column="RELATION_LEVEL"/>
            <field name="meetingLocation" column="MEETING_LOCATION"/>
        </class>
    </package>
</jdo>
```

So we've used a 1-N "CompoundIdentity" relation between *Customer* and *BusinessRelation*, and similarly between *Supplier* and *BusinessRelation* meaning that *BusinessRelation* has a composite PK define like this

```
public class BusinessRelation
{
    ...

    public static class PK implements Serializable
    {
        public LongIdentity customer; // Use same name as BusinessRelation field
        public LongIdentity supplier; // Use same name as BusinessRelation field

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer st = new StringTokenizer(s, "::");
            this.customer = new LongIdentity(Customer.class, st.nextToken());
            this.supplier = new LongIdentity(Supplier.class, st.nextToken());
        }

        public String toString()
        {
            return (customer.toString() + "::" + supplier.toString());
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }

        public int hashCode()
        {
            return customer.hashCode() ^ supplier.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.customer.equals(otherPK.customer) &&
this.supplier.equals(otherPK.supplier);
            }
            return false;
        }
    }
}
```

This arrangement will result in the following schema

So all we need to do now is persist some objects using these classes

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("jpox.properties");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
Object holderId = null;
try
{
    tx1.begin();

    Customer cust1 = new Customer("Web design Inc");
    Supplier supp1 = new Supplier("DataNucleus Corporation");
    pm.makePersistent(cust1);
    pm.makePersistent(supp1);

    BusinessRelation rel_1_1 = new BusinessRelation(cust1, supp1, "Very Friendly",
"Hilton Hotel, London");
    cust1.addRelation(rel_1_1);
    supp1.addRelation(rel_1_1);
    pm.makePersistent(rel_1_1);

    tx.commit();
}
finally
{
    if (tx1.isActive())
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    {
        tx1.rollback();
    }
    pm1.close();
}
```

This will now have persisted an entry in table "CUSTOMER", an entry in table "SUPPLIER", and an entry in table "BUSINESSRELATION". We can now utilise the *BusinessRelation* objects to update the attributes of the M-N relation as we wish.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.7 Datastore Replication

Data Replication

DataNucleus allows persistence of data into a datastore. In some situations an application needs to communicate with multiple datastores and so a form of data *replication* is required. DataNucleus allows this by use of the **attach/detach** functionality. We demonstrate this with an example

```
public class ElementHolder
{
    long id;
    private Set elements = new HashSet();

    ...
}

public class Element
{
    String name;

    ...
}

public class SubElement extends Element
{
    double value;

    ...
}
```

so we have a 1-N unidirectional (Set) relation, and we define the metadata like this

```
<jdo>
    <package name="org.jpox.test">
        <class name="ElementHolder" identity-type="application" detachable="true">
            <inheritance strategy="new-table"/>
            <field name="id" primary-key="true"/>
            <field name="elements" persistence-modifier="persistent">
                <collection element-type="org.jpox.test.Element"/>
                <join/>
            </field>
        </class>

        <class name="Element" identity-type="application" detachable="true">
            <inheritance strategy="new-table"/>
            <field name="name" primary-key="true"/>
        </class>

        <class name="SubElement">
            <inheritance strategy="new-table"/>
            <field name="value"/>
        </class>
    </package>
</jdo>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

and so in our application we create some objects in *datastore1*, like this

```
PersistenceManagerFactory pmf1 =
JDOHelper.getPersistenceManagerFactory("jpox.1.properties");
PersistenceManager pm1 = pmf1.getPersistenceManager();
Transaction tx1 = pm1.currentTransaction();
Object holderId = null;
try
{
    tx1.begin();

    ElementHolder holder = new ElementHolder(101);
    holder.addElement(new Element("First Element"));
    holder.addElement(new Element("Second Element"));
    holder.addElement(new SubElement("First Inherited Element"));
    holder.addElement(new SubElement("Second Inherited Element"));
    pm1.makePersistent(holder);

    tx1.commit();
    holderId = JDOHelper.getObjectId(holder);
}
finally
{
    if (tx1.isActive())
    {
        tx1.rollback();
    }
    pm1.close();
}
```

and now we want to replicate these objects into *datastore2*, so we detach them from *datastore1* and attach them to *datastore2*, like this

```
// Detach the objects from "datastore1"
ElementHolder detachedHolder = null;
pm1 = pmf1.getPersistenceManager();
tx1 = pm1.currentTransaction();
try
{
    pm1.getFetchPlan().setGroups(new String[] {FetchPlan.DEFAULT, FetchPlan.ALL});
    pm1.getFetchPlan().setMaxFetchDepth(-1);

    tx1.begin();

    ElementHolder holder = (ElementHolder) pm1.getObjectById(holderID);
    detachedHolder = (ElementHolder) pm1.detachCopy(holder);

    tx1.commit();
}
finally
{
    if (tx1.isActive())
    {
        tx1.rollback();
    }
    pm1.close();
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
// Attach the objects to datastore2
PersistenceManagerFactory pmf2 =
JDOHelper.getPersistenceManagerFactory("jpox.2.properties");
PersistenceManager pm2 = pmf2.getPersistenceManager();
Transaction tx2 = pm2.currentTransaction();
try
{
    tx2.begin();

    pm2.makePersistent(detachedHolder);

    tx2.commit();
}
finally
{
    if (tx2.isActive())
    {
        tx2.rollback();
    }
    pm2.close();
}
```

That's all there is. These objects are now replicated into *datastore2*. Clearly you can extend this basic idea and replicate large amounts of data.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.8 Spatial Types Tutorial

Persistence of Spatial Data

Background

DataNucleus-Spatial allows the use of DataNucleus as persistence layer for geospatial applications in an environment that supports the OGC SFA specification. It allows the persistence of the Java geometry types from the JTS topology suite as well as those from the PostGIS project.

In this tutorial, we perform the basic persistence operations over spatial types using Postgres and Postgis products.

1. Step 1 : Install the database server and spatial extensions.
2. Step 2 : Download DataNucleus and PostGis libraries.
3. Step 3 : Design and implement the persistent data model.
4. Step 4 : Design and implement the persistent code.
5. Step 5 : Run your application.

Step 1 : Install the database server and spatial extensions

Download Postgresql database and Postgis. Install Postgress and PostGis. During PostGis installation, you will be asked to select the database schema where the spatial extensions will be enabled. You will use this schema to run the tutorial application.

Step 2 : Download DataNucleus and PostGis libraries

Download the DataNucleus Core, RDBMS and Spatial libraries and any dependencies. Configure your development environment by adding the PostGis and JDO2/JPA1 jars to the classpath.

Step 3 : Design and implement the persistent data model

```
package org.datanucleus.samples.spatial.tutorial;

import org.postgis.Point;

public class Position
{
    private String name;

    private Point point;

    public Position(String name, Point point)
    {
        this.name = name;
        this.point = point;
    }
```

```
    public String getName()
    {
        return name;
    }

    public Point getPoint()
    {
        return point;
    }

    public String toString()
    {
        return "[name] "+ name + " [point] "+point;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
        <package name="org.datanucleus.samples.spatial.tutorial">

                <extension vendor-name="datanucleus" key="spatial-dimension"
value="2"/>
                <extension vendor-name="datanucleus" key="spatial-srid"
value="4326"/>

                <class name="Position" table="spatialpostut" detachable="true">
                        <field name="name"/>
                        <field name="point" persistence-modifier="persistent"/>
                </class>
        </package>
</jdo>
```

The above JDO metadata has two extensions *spatial-dimension* and *spatial-srid*. These settings specifies the format of the spatial data. *SRID* stands for spatial referencing system identifier and *Dimension* the number of coordinates.

Step 4 : Design and implement the persistent code

In this tutorial, we query for all locations where the distance is lower than 12 to the home position.

```
package org.datanucleus.samples.spatial.tutorial;

import java.sql.SQLException;
import java.util.List;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;
import javax.jdo.Transaction;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
import org.postgis.Point;

public class Main
{
    public static void main(String args[]) throws SQLException
    {
        // Create a PersistenceManagerFactory for this datastore
        PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("datanucleus.properties");

        System.out.println("DataNucleus Spatial Tutorial");
        System.out.println("====================");

        // Persistence of a Product and a Book.
        PersistenceManager pm = pmf.getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
        {
            //create objects
            tx.begin();

            Position[] sps = new Position[3];
            Point[] points = new Point[3];
            points[0] = new Point("SRID=4326;POINT(5 0)");
            points[1] = new Point("SRID=4326;POINT(10 0)");
            points[2] = new Point("SRID=4326;POINT(20 0)");
            sps[0] = new Position("market",points[0]);
            sps[1] = new Position("rent-a-car",points[1]);
            sps[2] = new Position("pizza shop",points[2]);
            Point homepoint = new Point("SRID=4326;POINT(0 0)");
            Position home = new Position("home",homepoint);

            System.out.println("Persisting spatial data...");
            System.out.println(home);
            System.out.println(sps[0]);
            System.out.println(sps[1]);
            System.out.println(sps[2]);
            System.out.println("");

            pm.makePersistentAll(sps);
            pm.makePersistent(home);

            tx.commit();

            //query for the distance
            tx.begin();

            Double distance = new Double(12.0);
            System.out.println("Retriving position where distance to home is less
than "+distance+" ... Found:");

            Query query = pm.newQuery(Position.class, "name != 'home' &&
Spatial.distance(this.point, :homepoint) < :distance");
            List list = (List) query.execute(homepoint, distance);
            for( int i=0; i<list.size(); i++)
            {
                System.out.println(list.get(i));
            }
            //clean up database.. just for fun :)
            pm.newQuery(Position.class).deletePersistentAll();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }

        System.out.println("");
        System.out.println("End of Tutorial");
    }
}
```

We define the file with connection properties to Postgresql:

```
#database host / ip / database
javax.jdo.option.ConnectionDriverName=org.postgresql.Driver
javax.jdo.option.ConnectionURL=jdbc:postgresql://localhost:5432/postgis
javax.jdo.option.ConnectionURL2=jdbc:postgresql://localhost:5432/postgis
javax.jdo.option.ConnectionUserName=postgres
javax.jdo.option.ConnectionPassword=postgres

javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.Optimistic=false
javax.jdo.option.IgnoreCache=false
datanucleus.autoCreateTables=true
datanucleus.validateTables=true
datanucleus.autoCreateColumns=true
datanucleus.autoCreateConstraints=true
datanucleus.validateConstraints=true
datanucleus.autoCreateSchema=true
```

Step 5 : Run your application

Before running the application, you must enhancethe persistent classes. Finally, configure the application classpath with the DataNucleus Core, DataNucleus RDBMS, DataNucleus Spatial, JDO2, Postgres and PostGis libraries and run the application as any other java application.

The output for the application is:

```
Output :

DataNucleus Spatial Tutorial
=====================
Persisting spatial data...
[name] home [point] SRID=4326;POINT(0 0)
[name] market [point] SRID=4326;POINT(5 0)
[name] rent-a-car [point] SRID=4326;POINT(10 0)
[name] pizza shop [point] SRID=4326;POINT(20 0)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Retriving position where distance to home is less than 12.0 ... Found:
[name] market [point] SRID=4326;POINT(5 0)
[name] rent-a-car [point] SRID=4326;POINT(10 0)

End of Tutorial
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.9 JFire

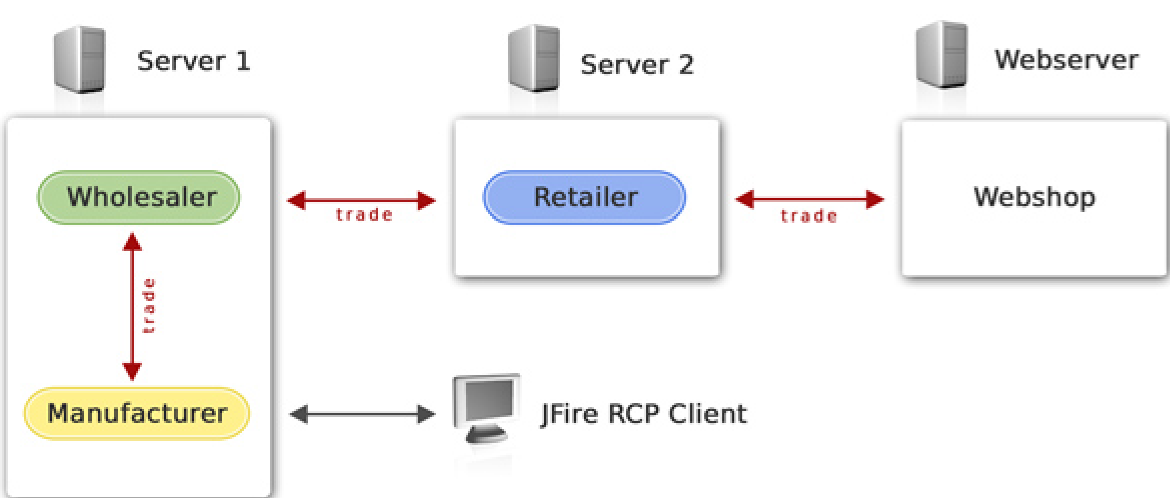
JFire

JFire is a new, powerful and free ERP, CRM, eBusiness and SCM/SRM solution for business enterprises. JFire is entirely free/open-source software, uses the latest technologies (J2EE 1.4, JDO 2.0, Eclipse RCP 3.2) and is designed to be highly customizable.

It is of interest for DataNucleus users because it is a good example of use of JDO 2.0 in a real-world application, and is developed using DataNucleus. Being open-source you can view how JDO 2.0 is employed to provide its data persistence. It is a prime example of usage of XDoclet 1.2.* JDO2 tags, too.

Basic setup

Being a client-server system, JFire uses a J2EE server for the backend, into which Enterprise Archives (EARs) are deployed. These EARs contain Resource Adapters, (stateless) Session Beans and the data model. Entity Beans are not used, because the data model is pure JDO 2.0. Even though J2EE allows the easy implementation of all kinds of clients, there is so far solely an Eclipse-RCP-based client implemented.



In order to guarantee strict privacy protection, JFire manages every organisation in a separate datastore. The organisations can even be hosted on different J2EE servers and thus be spread around the world. This makes the system highly scalable - in most cases even without employing a complex clustering configuration.

Because JFire is built in a very modular form, parts of it can be used for totally different purposes (e.g. solely for organisation and user management, without any trade related functionality).

XDoclet

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Having written JDO metadata, you've probably recognized that you have to do some duplicate work - for example the names of the fields occur in both, the java classes and the XML files. For small projects, this is really not an issue, but when having more than a hundred JDO classes, you probably prefer to reduce duplicate work wherever possible. Additionally, maintaining two different files describing the same object is more error-prone than having all information for one class gathered in one single file.

Therefore, JFire makes heavy use of XDoclet for the auto-generation of JDO meta data, objectid-classes (see application identity) and EJB interfaces/util-classes.

Note, that the official XDoclet release very likely does not yet include support for all JDO 2.0 features. You should either use a recent CVS snapshot binary or build it yourself from current sources.

If you want to see how a JDO-tagged java class looks like, you can find an interesting example here (direct link to source code). This abstract class uses application identity with a composite primary key. It declares queries, fetch-groups and much more.

Cache for the client

DataNucleus has already some caching functionality included (see Cache and Cache (Developer)). These caches don't span the client however. That's why JFire implements a cache system which tracks changes to JDO objects in the server and is able to notify listeners of these changes. The JFire RCP-client uses this to cache detached JDO objects and reduce queries. This is part of the basic framework (JFireBase) and could be used in any other application that works on a J2EE server with JDO data-model and RCP clients.

Inter-Organisation-Sync

JFire's JDO-cache-brigde is also used for another fundamental feature of JFire's base: the synchronisation of objects between organisations in a JFire network. It registers listeners to the cache and informs synchronised candidates about changes. Being push-based, remote organisations are instantly notified. The receiver however is himself responsible to actually grab the changed data and can independently decide when and to what extent this should be done.

BIRT datasource for JDO

BIRT is an open source reporting framework that JFire uses for creation and rendering of forms like invoices or delivery notes or to generate statistical reports. The project JFireReporting provides a new data-source for BIRT reports that let users access a PersistenceManager in the JFire server. Users can collect their data either by JavaScript with access to a PersistenceManager or via direct JDOQL. Take a look at the documentation for JFireReporting or browse the source code for more information.

Download

The only thing remaining is to download JFire now and see how you can utilise JDO 2 and DataNucleus in your application. You might want to use parts of JFire, even if you don't implement a trade-related application, because user-management, remote-class-loading, client-sided caching and other functionality is helpful to many other use-cases, too.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Authors: Alexander Bieber, Marco Schulze

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.10 Springframework

DataNucleus and the SpringFramework

The Spring Framework provides a mechanism for designing systems, aiding you in the modularisation of components and hence in making components more readily testable. The crux of the framework is that you design your business service and data access objects as Java beans, and then provide a dependency mapping between them. This leads to very well structured systems with a pluggable feel.

The most important thing to mention is that you must use Spring 2.0 or later.

Spring Framework is suited to a wide variety of architectures, and can be utilised in discrete parts of a system, as well as across the whole system. Let's give an example where we want to use Spring for the business and data access tiers of a system.

In our system we have a typical business service SampleService, which utilises a data-access SampleDAO. We define these as Java Beans (default constructor, and properties with getters/setters). Once we have defined our beans we then define the "glue" to link them together. This is performed via an XML configuration file. There are many ways you can utilise Spring in this respect. Here's our definition using what is typically the simplest pattern. This file is used to create an ApplicationContext which loads all of these beans at startup automatically.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- PMF Bean -->
    <bean id="pmf"
        class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
        <property name="jdoProperties">
            <props>
                <prop key="javax.jdo.PersistenceManagerFactoryClass">
                    org.jpox.jdo.JDOPersistenceManagerFactory</prop>
                <prop
key="javax.jdo.option.ConnectionURL">jdbc:mysql://localhost/dbname</prop>
                <prop key="javax.jdo.option.ConnectionUserName">username</prop>
                <prop key="javax.jdo.option.ConnectionPassword">password</prop>
                <prop
key="javax.jdo.option.ConnectionDriverName">com.mysql.jdbc.Driver</prop>
            </props>
        </property>
    </bean>

    <!-- Transaction Manager for PMF -->
    <bean id="jdoTransactionManager"
class="org.springframework.orm.jdo.JdoTransactionManager">
        <property name="persistenceManagerFactory">
            <ref local="pmf"/>
        </property>
    </bean>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <!-- Typical DAO -->
    <bean id="sampleDAO" class="org.jpox.spring.SampleDAO">
        <property name="persistenceManagerFactory">
            <ref local="pmf"/>
        </property>
    </bean>

    <!-- Typical Business Service -->
    <bean id="sampleService" class="org.jpox.spring.SampleService">
        <property name="sampleDAO">
            <ref local="sampleDAO"/>
        </property>
    </bean>

    <!-- Transaction Interceptor for Business Services -->
    <bean id="transactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref local="jdoTransactionManager">
        </property>
        <property name="target">
            <ref local="sampleService">
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="store*">PROPAGATION_REQUIRED</prop>
                <prop key="delete*">PROPAGATION_REQUIRED</prop>
                <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
            </props>
        </property>
    </bean>
</beans>
```

Here we've defined our SampleService to have a dependency on SampleDAO. In turn, SampleDAO has a dependency on the PersistenceManagerFactory. We've opted to use Spring's transaction handling capability, so we define a JdoTransactionManager for the PersistenceManagerFactory and in addition, we define a transaction interceptor around the SampleService which couples our transactions with those of Spring.

Design of a JDO DAO

The design of your data persistence layer should be the only place you actually interact with your JDO implementation (e.g DataNucleus). Outside of the DAO your Java objects should be just that, Java objects. This allows you to delimit the impact of your choice of data persistence, and leaves a clean interface to the rest of your system. It provides you the flexibility to swap in a new data persistence strategy in the future.

The choice of JDO for data persistence implies certain operations within this layer so as to provide your clean interface. The JDO 2.0 specification provides 2 important changes to make this simpler, namely the attach/detach capability, and the use of fetch-groups. To give an example of a sample DAO using JDO, ...

ALL RIGHTS RESERVED

```
public class SampleDAO extends JdoDaoSupport
{
    /** Accessor for a collection of objects */
    public Collection getWorkers()
    throws DataAccessException
    {
        Collection workers = getJdoTemplate().find(Worker.class, null,
            "lastName ascending");
        workers = getPersistenceManager().detachCopyAll();
        return workers;
    }

    /** Accessor for a specified object */
    public Worker loadWorker(long id)
    throws DataAccessException
    {
        Worker worker =
            (Worker)getJdoTemplate().getObjectById(Worker.class, new Long(id));
        if (worker == null)
        {
            throw new RuntimeException("Worker " + id + " not found");
        }
        return (Worker) getPersistenceManager().detachCopy(worker);
    }

    /** Save/Update an object */
    public void storeWorker(Worker worker)
    throws DataAccessException
    {
        getJdoTemplate().makePersistent(worker);
    }

    /** Delete an object. */
    public void deleteWorker(Worker worker)
    throws DataAccessException
    {
        if (worker == null || worker.getId() == null)
        {
            throw new RuntimeException("Worker is not persistent");
        }
        else
        {
            getPersistenceManager().deletePersistent(worker);
        }
    }
}
```

The example above demonstrates the 4 most common types of data accessor methods ... namely retrieval of a collection of objects based on a query, load of an object given an id, the saving/updating of an object, and the deletion of the object. Use of the Spring JdoTemplate provides wrapping of the majority of JDO methods required by an application. The DAO extends the Spring JdoDaoSupport class, giving it the benefits of Spring's facilities. The use of attach/detach is highlighted above. Whenever an object is retrieved from the datastore and needs to be used in the application, it is detached from the persistence graph using the detachCopy/detachCopyAll methods. Whenever an object needs persisting, if it is new then it is persisted directly, and otherwise it is reattached to the persistence graph and the datastore updated with any changes.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The other aspect to note is where you have relationships between your Java objects. In this case you need to define a **fetch-group** defining which objects in a relationship are loaded when you detach the objects from the persistence graph. That is, which related objects you will be needing to use within your application. This is specified in the JDO Meta-Data for your Java classes.

DataNucleusClinic

We now demonstrate the above techniques in a real application. Here we take the Spring Framework example of **Petclinic** and adapt it to use with a JDO persistence layer (here we use DataNucleus, naturally!).

Welcome Find owner Display owners Display veterinarians JPOX-Spring Tutorial Petclinic Tutorial

Welcome to the JPOXClinic

JPOXClinic provides an introduction to using the SpringFramework with the JPOX JDO implementation. It demonstrates the use of Spring transaction management, and how to integrate the persistence code in a typical DAO layer with the business and web tiers, accessing the persisted objects for display and update.

JPOXClinic :: a Spring Framework JPOX demonstration

Changes to Petclinic

In working to make Petclinic work with DataNucleus, it was necessary to make some minor changes to the original Spring codebase. These are detailed below.

- The data model classes are now moved to a package model to better separate the components of the system. In addition the DAO interface and its JDO implementation are in a package dao.
- The Petclinic build used Ant, and had no provision for byte-code enhancement of the model classes. The build provided with DataNucleusClinic uses Maven and has the enhancement included. This happens automatically when you compile.
- DataNucleus provides the facility to create the database schema automatically at runtime, or via the SchemaTool. As a result we don't provide schema installation scripts, just a populate script.

Model Configuration

In DataNucleusClinic we have the following persisted classes

- Person - superclass of an Owner/Vet
- Owner - The owner of a collection of Pets.
- Pet - An animal with an owner (1-N relationship). Has a PetType. Has a collection of Visits to the Vet
- PetType - Type of a Pet

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- Vet - A person who repairs Pets when they need it. Has a collection of Specialtys.
- Visit - A visit of a Pet to a Vet
- Specialty - A category of veterinary science that a Vet specialises in
- Entity - base class that provides an identity.
- NamedEntity - extension of Entity adding a name.

In our JDO Meta-Data we define which fields are to be persisted, which fields relate to which other objects, and how we will retrieve these related objects (using fetch-groups). You will see that the Entity and NamedEntity classes don't have their own tables.

```
<jdo>
    <package name="org.jpox.samples.jpoxclinic.model">

        <class name="Entity" detachable="true" identity-type="application">
            <inheritance strategy="subclass-table"/>
            <field name="id" primary-key="true" value-strategy="identity"/>
        </class>

        <class name="NamedEntity" detachable="true" identity-type="application">
            <inheritance strategy="subclass-table"/>
            <field name="name" column="name">
                <column length="80" jdbc-type="VARCHAR"/>
            </field>
        </class>

        <!-- Person Class. Map to table "person" (subclasses) -->
        <class name="Person" detachable="true" table="person">
            <inheritance strategy="new-table"/>
            <field name="firstName">
                <column length="30" jdbc-type="VARCHAR"/>
            </field>
            <field name="lastName">
                <column length="30" jdbc-type="VARCHAR"/>
            </field>
            <field name="address">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="city">
                <column length="80" jdbc-type="VARCHAR"/>
            </field>
            <field name="telephone">
                <column length="20" jdbc-type="VARCHAR"/>
            </field>
        </class>

        <!-- Owner Class. Map to table "owners" and provide fetch-group -->
        <class name="Owner" detachable="true" table="owners"
identity-type="application">
            <inheritance strategy="new-table"/>
            <field name="pets" mapped-by="owner">
                <collection element-type="org.jpox.samples.jpoxclinic.model.Pet"/>
            </field>
            <fetch-group name="detach_owner_pets">
                <field name="pets"/>
            </fetch-group>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <!-- Vet Class. Map to table "vets" and provide fetch-group -->
        <class name="Vet" detachable="true" table="vets"
identity-type="application">
            <inheritance strategy="new-table"/>
            <field name="specialties" table="vet_specialties">
                <collection
element-type="org.jpox.samples.jpoxclinic.model.Specialty"/>
                <join>
                    <column name="vet_id"/>
                </join>
                <element>
                    <column name="specialty_id"/>
                </element>
            </field>
            <fetch-group name="detach_vet_specialties">
                <field name="specialties"/>
            </fetch-group>
        </class>

        <!-- Specialty Class. Map to table "specialties" and provide fetch-group -->
        <class name="Specialty" detachable="true" table="specialties"
identity-type="application">
            <inheritance strategy="new-table"/>
            <field name="vets" table="vet_specialties">
                <collection element-type="org.jpox.samples.jpoxclinic.model.Vet"/>
                <join>
                    <column name="specialty_id"/>
                </join>
                <element>
                    <column name="vet_id"/>
                </element>
            </field>
            <fetch-group name="detach_specialty_vets">
                <field name="vets"/>
            </fetch-group>
        </class>

        <!-- PetType Class. Map to table "types" -->
        <class name="PetType" detachable="true" table="types"
identity-type="application">
            <inheritance strategy="new-table"/>
        </class>

        <!-- Pet Class. Map to table "pets" -->
        <class name="Pet" detachable="true" table="pets"
identity-type="application">
            <inheritance strategy="new-table"/>
            <field name="birthDate" column="birth_date"/>
            <field name="owner" column="owner_id"
persistence-modifier="persistent"/>
            <field name="type" column="type_id" persistence-modifier="persistent"/>
            <field name="visits" mapped-by="pet">
                <collection element-type="org.jpox.samples.jpoxclinic.model.Visit"/>
            </field>

            <fetch-group name="detach_pet">
                <field name="type"/>
            </fetch-group>
            <fetch-group name="detach_pet_owner">
                <field name="owner"/>
            </fetch-group>
            <fetch-group name="detach_pet_visits">
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
                <field name="visits"/>
            </fetch-group>
        </class>

        <class name="Visit" detachable="true" table="visits"
identity-type="application">
            <inheritance strategy="new-table"/>
            <field name="date" column="visit_date"/>
            <field name="description" column="description">
                <column length="255" jdbc-type="VARCHAR"/>
            </field>
            <field name="pet" column="pet_id" persistence-modifier="persistent"/>

            <fetch-group name="detach_visit_pet">
                <field name="pet"/>
            </fetch-group>
        </class>
    </package>
</jdo>
```

The other thing to note from this MetaData is the use of 2 other new features of JDO 2. The first is called SingleFieldIdentity and means that where we have only a single primary key field, we no longer need to provide a primary key class, and so we miss off the objectid-class attribute. The second feature is part of the JDO 2 O/R mapping definition. You note that we have 2 classes Entity and NamedEntity and we don't want these to have their own tables in the database. We simply define them as having an inheritance strategy of subclass-table. This means that their fields will be persisted in the table of the next subclass that does have its own table.

Data Access Object Configuration

As mentioned in the description of how to design a dataaccess layer with JDO we need to make use of the JDO 2.0 features attach/detach and fetch-groups. With DataNucleusClinic we use this strategy. Lets take the example of a finder for the Owner class.

```
    public Collection findOwners(String lastName) throws DataAccessException
    {
        getPersistenceManager().getFetchPlan().addGroup("detach_owner_pets");
        getPersistenceManager().getFetchPlan().addGroup("detach_pet");
        getPersistenceManager().getFetchPlan().addGroup("detach_pet_visits");
        getPersistenceManager().getFetchPlan().setMaxFetchDepth(4);
        Map values = new HashMap();
        values.put("value",lastName);
        Collection owners =
            getJdoTemplate().find(Owner.class, "lastName == value","String
value",values);
        owners = getPersistenceManager().detachCopyAll(owners);
        return owners;
    }
```

So we define that when we fetch our Owner objects, we retrieve the "pets" related objects, and for each of them we retrieve the "petType", and the "visits" for each Pet. We then detach the Owner objects (and with them the Pet objects) so that we can use them in our application.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

This demonstrates the power of the fetch-group and attach/detach functionality. If you look at the other methods in the DataNucleusClinic DAO you will find a similar methodology applied throughout.

Spring Configuration

Lets look at how we configure Spring in terms of dependency injection. We define our DataNucleusClinic class as the DAO bean, having a dependency on the PersistenceManagerFactory, and the PersistenceManagerFactory taking the standard DataNucleus properties for the datastore and to auto-generate the datastore schema (if not already existing). Finally we define Spring's transaction interceptor to provide all transaction handling so we don't have to write the code for this ourselves.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- DataNucleus PersistenceManagerFactory -->
    <bean id="pmf"
class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
        <property name="jdoProperties">
            <props>
                <prop key="javax.jdo.PersistenceManagerFactoryClass">
                    org.jpox.jdo.JDOPersistenceManagerFactory</prop>
                <prop
key="javax.jdo.option.ConnectionURL">jdbc:mysql://localhost/petclinic</prop>
                <prop key="javax.jdo.option.ConnectionUserName">pc</prop>
                <prop key="javax.jdo.option.ConnectionPassword"></prop>
                <prop
key="javax.jdo.option.ConnectionDriverName">com.mysql.jdbc.Driver</prop>
                <prop key="org.jpox.autoCreateSchema">true</prop>
                <prop key="org.jpox.identifier.case">PreserveCase</prop>
            </props>
        </property>
    </bean>

    <!-- Transaction manager for a single DataNucleus PMF -->
    <bean id="transactionManager"
class="org.springframework.orm.jdo.JdoTransactionManager">
        <property name="persistenceManagerFactory"><ref local="pmf"/></property>
    </bean>

    <!-- DataNucleusClinic primary DAO -->
    <bean id="clinicTarget"
class="org.jpox.samples.jpoxclinic.dao.DataNucleusClinic">
        <property name="persistenceManagerFactory"><ref local="pmf"/></property>
    </bean>

    <!-- Transactional proxy for the DataNucleusClinic primary DAO -->
    <bean id="clinic"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref
local="transactionManager"/></property>
        <property name="target"><ref local="clinicTarget"/></property>
        <property name="transactionAttributes">
            <props>
                <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
                <prop key="store*">PROPAGATION_REQUIRED</prop>
                <prop key="delete*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
</beans>
```

The final part of this defines which methods Spring manages the transactions for, and how it manages them. So it will manage transactions for all methods starting "get", "find", "load", "store", and "delete"

Summary

From this brief tutorial you have seen how we have designed a simple data model and how we configured it for data persistence by DataNucleus. We then defined a data access layer for our application, and finally configured our data access layer and model providing dependency injection and transactions, giving us a very flexible application structure. We have only really touched on some of Spring's capabilities, and you should refer to the Spring Framework project for its full capabilities. You can download the DataNucleusClinic sample application from SourceForge.

Much of the above example code uses "JdoTemplate" which wraps JDO methods. It is arguable as to the benefit of this and indeed the fact that SpringFramework is sometimes behind JDO in terms of support we recommend using Spring for providing the transaction handling, and then just use JdoTemplate to get hold of the PersistenceManager, thereafter using JDO methods directly.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.11 DAO Layer Design

DataNucleus - Design of a DAO Layer with JDO

Introduction

The design of an application will involve many choices, and often compromises. What is generally accepted as good practice is to layer the application tiers and provide interfaces between these. For example a typical web application will generally have 3 tiers - web tier, business-logic tier, and data-access tier. DataNucleus provides data persistence and so, following this model, should only be present in the data access layer. This layer is generally designed using the data access object (DAO) pattern.

A typical DAO provides an interface that defines its contract with the outside world. This takes the form of a series of data access and data update methods. In this tutorial we follow this and describe how to implement a DAO using DataNucleus and JDO using some of the new features introduced in JDO 2.0

The DAO contract

To highlight our strategy for DAO's we introduce 3 simple classes.

```
public class Owner
{
    private Long id;
    private String firstName;
    private String lastName;
    private Set pets;

    public void addPet(Pet pet)
    {
        pets.add(pet);
    }
    ...
}
public class Pet
{
    private Long id;
    private PetType type;
    private String name;
    private Owner owner;

    public void setType(PetType type)
    {
        this.type = type;
    }
    ...
}

public class PetType
{
    private String name;
    ...
}
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So we have 3 dependent classes, and we have a 1-N relationship between Owner and Pet, and a N-1 relationship between Pet and PetType.

We now generate an outline DAO object containing the main methods that we expect to need

```
public interface ClinicDAO
{
    public Collection getOwners();
    public Collection getPetTypes();
    public Collection findOwners(String lastName);
    public Owner loadOwner(long id);
    public void storeOwner(Owner owner);
    public void storePet(Pet pet);
}
```

Clearly we could have defined more methods, but these will demonstrate the basic operations performed in a typical application. Note that we defined our DAO as an interface. This has various benefits, and the one we highlight here is that we can now provide an implementation using DataNucleus and JDO. We could, in principle, provide a DAO implementation of this interface using JDBC for example, or one for whatever persistence technology. It demonstrates a flexible design strategy allowing components to be swapped at a future date.

We now define an outline DAO implementation using DataNucleus. We will implement just a few of the methods defined in the interface, just to highlight the style used. So we choose one method that retrieves data and one that stores data.

```
public class MyDAO implements ClinicDAO
{
    PersistenceManagerFactory pmf;

    /** Constructor, defining the PersistenceManagerFactory to use. */
    public MyDAO(PersistenceManagerFactory pmf)
    {
        this.pmf = pmf;
    }

    /** Accessor for a PersistenceManager */
    protected PersistenceManager getPersistenceManager()
    {
        return pmf.getPersistenceManager();
    }

    public Collection getOwners()
    {
        Collection owners;
        PersistenceManager pm=getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
        {
            tx.begin();
            Query q=pm.newQuery(org.jpox.samples.jpoxclinic.model.Owner.class);
            Collection query_owners=q.execute();

            // *** TODO Copy "query_owners" into "owners" ***
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
        return owners;
    }

    public void storeOwner(Owner owner)
    {
        PersistenceManager pm=getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
        {
            tx.begin();
            // Owner is new, so persist it
            if (owner.id() == null)
            {
                pm.makePersistent(owner);
            }
            // Owner exists, so update it
            else
            {
                // *** TODO Store the updated owner ***
            }

            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
    }

    ...
}
```

So here we've seen the typical DAO and how, for each method, we retrieve a PersistenceManager, obtain a transaction, and perform our operation(s). Notice above there are a couple of places where we have left "TODO" comments. These will be populated in the next section, using the JDO 2.0 feature attach/detach.

Use of attach/detach

We saw in the previous section our process for the DAO methods. The problem we have with JDO 1.0 is that as soon as we leave the transaction our object would move back to "Hollow" state (hence losing its field values, and hence the object would have been unusable in the rest of our application. With JDO 2.0

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

we have a feature called attach/detach that allows us to detach objects for use elsewhere, and then attach them when we want to persist any changed data within the object. So we now go back to our DataNucleus DAO and add the necessary code to use this

```
    public Collection getOwners()
    {
        Collection owners;
        PersistenceManager pm=getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
        {
            tx.begin();
            Query q=pm.newQuery(org.jpox.samples.jpoxclinic.model.Owner.class);
            Collection query_owners=q.execute();

            // Detach our owner objects for use elsewhere
            owners = pm.detachCopyAll(query_owners);

            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
        return owners;
    }

    public void storeOwner(Owner owner)
    {
        PersistenceManager pm=getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
        {
            tx.begin();

            // Persist our changes back to the datastore
            pm.makePersistent(owner);

            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
    }
}
```

So we have added 2 very simple method calls. These facilitate making our objects usable outside the DAO layer and so give us much flexibility. Please note that instead of *detachCopy* you could set the PMF

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

option "javax.jdo.option.DetachAllOnCommit" and this would silently migrate all enlisted instances to detached state at commit of the transaction, and is probably a more convenient way of detaching.

Definition of fetch-groups

In the previous section we have described how to design our basic DAO layer. We have an interface to this layer and provide a DataNucleus implementation. The DataNucleus/JDO calls are restricted to the DAO layer. What we haven't yet considered is what actually is made usable in the rest of the application when we do our detach. By default with this feature PersistenceCapable fields will not be detached with the owning object. This means that our "pets" field of our detached "owner" object will not be available for use. In many situations we would want to give access to other parts of our object. To do this we make use of another JDO 2.0 feature, called fetch-groups. This is defined both in the Meta-Data for our classes, and in the DAO layer where we perform the detaching. Let's start with the MetaData for our 3 classes.

```
<class name="Owner" detachable="true" identity-type="application"
    objectid-class="org.jpox.samples.jpoxclinic.model.key.OwnerKey">
    <field name="pets" mapped-by="owner">
        <collection element-type="org.jpox.samples.jpoxclinic.model.Pet"/>
    </field>
    <fetch-group name="detach_owner_pets">
        <field name="pets"/>
    </fetch-group>
</class>
<class name="PetType" detachable="true" identity-type="datastore">
    <field name="name">
        <column length="80" jdbc-type="VARCHAR"/>
    </field>
</class>
<class name="Pet" detachable="true" identity-type="application"
    objectid-class="org.jpox.samples.jpoxclinic.model.key.PetKey">
    <field name="id" primary-key="true" value-strategy="true"/>
    <field name="name">
        <column length="30" jdbc-type="VARCHAR"/>
    </field>
    <field name="type" persistence-modifier="persistent"/>
    <fetch-group name="detach_pet_type">
        <field name="type"/>
    </fetch-group>
</class>
```

Here we've marked the classes as detachable, and added a fetch-group to Owner for the "pets" field, and to Pet for the "type" field. Doing these for each field adds extra flexibility to our ability to specify them. Lets now update our DAO layer method using detach to use these fetch-groups so that when we use the detached objects in our application, they have the necessary components available.

```
    public Collection getOwners()
    {
        Collection owners;
        PersistenceManager pm=getPersistenceManager();
        Transaction tx=pm.currentTransaction();
        try
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        {
            tx.begin();

            // Define the objects to be fetched/detached with our owner objects.
            pm.getFetchPlan().addGroup("detach_owner_pets");
            pm.getFetchPlan().addGroup("detach_pet_type");
            pm.getFetchPlan().setMaxFetchDepth(3);

            Query q=pm.newQuery(org.jpox.samples.jpoxclinic.model.Owner.class);
            Collection query_owners=q.execute();

            // Detach our owner objects for use elsewhere
            owners = pm.detachCopyAll(query_owners);

            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
        return owners;
    }
```

So you see that when we detach our Owner objects, we also detach the Pet objects for each owner and for each Pet object we will also detach the PetType object. This means that we can access all of these objects with no problem outside of the DAO layer.

Summary

In this tutorial we've demonstrated a way of separating the persistence code from the rest of the application by providing a DAO layer interface. We've demonstrated how to write the associated methods within this layer and, with the use of attach/detach and fetch-groups we can make our persisted objects available outside of the DAO layer in a seamless way. Developers of the remainder of the system don't need to know any details of the persistence, they simply code to the interface contract provided by the DAO.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.12 Tapestry : VLib

JPOX and Tapestry : The VLib example

Document version 0.92.
© 2004 David Ezzio. All rights reserved.
This tutorial explains how to use JPOX with Tapestry, using the VLib example as the template. JPOX provides a downloadable sample with the code described here.

What is the VLib example ?

This example, called VLib, is a port of the Virtual Library example distributed with Apache Tapestry. It shows a Tapestry application using JDO.

The original Virtual Library example is an ambitious example intended to show off the capabilities of Tapestry, a Web application framework. The fictitious business case for the Virtual Library application is the notion of a book club, perhaps at work, where people can borrow each other's books. The Virtual Library keeps track of what books are available, who owns them, and who has borrowed them. There are three levels of access: anonymous, logged-in, and administrative. The example looks sharp with a heavy emphasis on graphics in the presentation. Howard Lewis Ship, the architect of Tapestry, gives a two chapter tutorial of the Virtual Library example in his book **Tapestry in Action**, published by Manning.

As written, the VLib port runs with JPOX 1.0.2 (and later) and JPOX 1.1.0.A2 and later. As of these JPOX versions, the same VLib example works with both versions. As JPOX 1.1 evolves, it is expected that the VLib version for JPOX 1.1 will evolve to make use of some of the new features in JDO 2.0.

Who is most likely to benefit from this example ?

Developers who are familiar with or learning Tapestry will benefit the most from this example.

What is Tapestry ?

Tapestry is a Web application framework from Apache. It can be found at http://jakarta.apache.org/tapestry.

Web applications within Tapestry follow the model-view-controller (MVC) pattern. The view consists of HTML templates, and specifications in XML for JavaScript, as well as style sheets and graphic resources. The controller consists of Java code with page and component wiring in XML. Tapestry does not provide a model, making it an ideal partner for JDO.

Tapestry abstracts from the multi-threadedness of the Servlet API and provides tight coupling between the HTTP request and response handling code. The same components that participate in the rendering of page also accept the user's response to the page. The handshaking between HTTP responses and the Java code is handled by the framework, so that most dynamic information presented on the page and most

user input to the page are represented as properties of the Tapestry page or its components. The user's next request action results in an invocation of a listener method on the page or one of its components. Every Tapestry page and component is single-threaded by design, and contrary to what you might be inclined to believe, it performs quite well because of object pooling.

How does the VLib example differ from the original?

The VLib example is a port (or fork) of the Virtual Library example that is distributed with Apache Tapestry.

Unlike the Virtual Library example which uses Enterprise JavaBeans to implement the model, the VLib example uses JDO.

In general, most of the presentation files are unchanged, and those that are changed, are changed very little. The controller code varies class-by-class from no change, to one or two little changes, to many small changes. The model code has been, for the most part, completely rewritten, although there is a family resemblance to the original code.

Where can I see VLib example working?

It would be useful to put the VLib example on the Web, but in the meantime, setting it up on your own machine is not hard.

How do I set up VLib in my development environment?

This tutorial assumes that you want to start by just deploying the example and seeing it work, before you dive into the source code and Ant build.

Install Tapestry 3.0. If you are just learning Tapestry, you should get the Hangman example working. The book **Tapestry in Action** from Manning Publications can be quite useful.

Install JPOX 1.1.0-alpha-2 (or later). If you are new to JDO, you should work through the JPOX tutorial. The book **Using and Understanding Java Data Objects** from Apress can be quite useful for understanding both the details of JDO and the big picture.

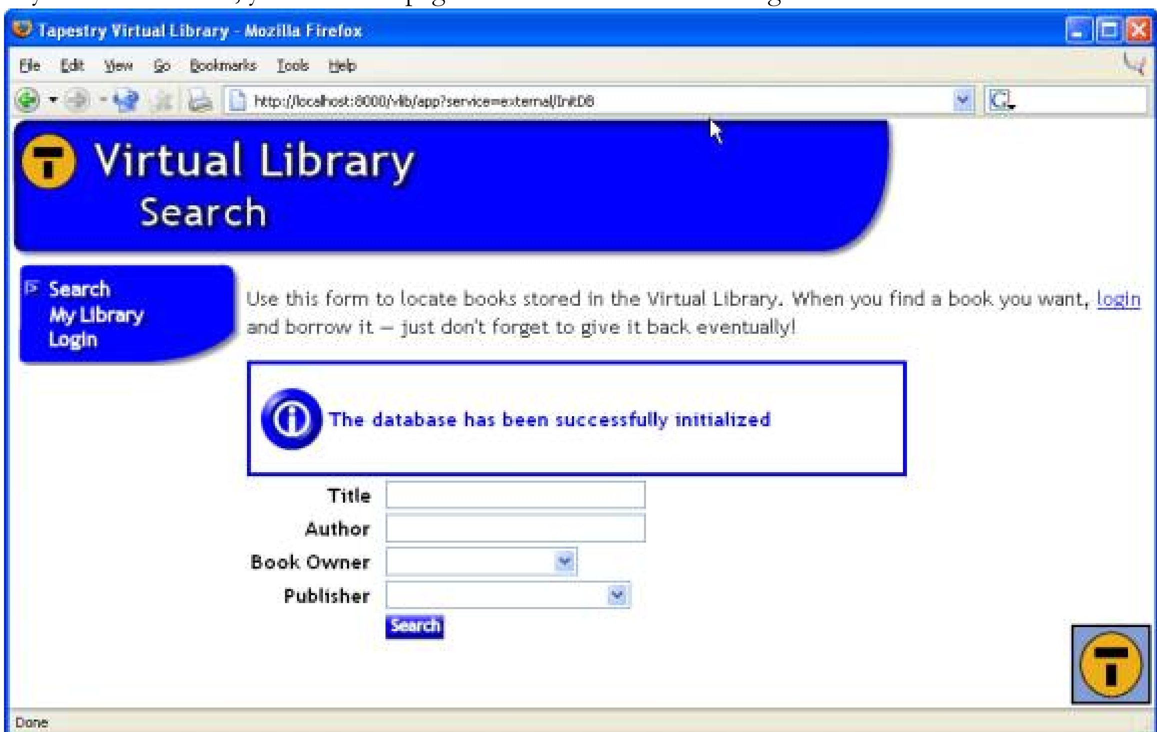
With the preliminaries taken care of, let's get started

1. Determine which JDBC driver JPOX needs to talk to your favorite (and JPOX supported) database. Put this driver into the appropriate place to make your favorite Web app container happy. For example, Tomcat wants it in the **<tomcat>/common/lib** directory.
2. Start your Web app container. For example, start up Tomcat.
3. Unzip the example binary distribution, and deploy the **vlib.war** file to your Web app container. For example, copy it to the **<tomcat>/webapps** directory. If the container does not unzip the War file for you, unzip it to a **vlib** directory under the appropriate directory for your container. For example, unzip it to **<tomcat>/webapps/vlib**.
4. Start your database server.
5. You may want to create the database where you want the VLib data to be stored. For example, I use MySQL, and I have a database named **vlib** where the VLib data is stored. If you have a general test

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

database that you use for lots of stuff, that should work as well.

6. You may want to create a user account for the VLib application. You can also use an existing account.
7. Find the **jdo.properties** file located in the unwrapped War file directory, **<webapps-root>\vlib\WEB-INF\classes** directory. Edit the four connection properties to connect to your database using the parameters for your JDBC driver. These connection properties provide the information that JPOX requires when it uses JDBC to connect to your database.
8. At this point, there is no schema in your database to hold the persistent state of the VLib's data objects, and there is no data for the objects either. So the next step, which should be done only once, will create the schema and add the testing data. If you initialize the data more than once, you'll see duplicate books. To remove the duplicates, drop all the tables that the initialization creates, and try again.
9. To initialize the database, use your Web browser to hit a URL that looks like the following: **http://localhost:8000/vlib/app?service=external/InitDB**
   You may have to alter the host name and the port to conform to your Web container's configuration.
10. If you are successful, you will see a page that looks like the following:



From this point on, you can just use the application. The book **Tapestry in Action** contains a tutorial for the user interface. There are four user accounts already defined. Suzy Queue is **squeue@bug.org**. Frodo Baggins is **ringbearer@bagend.shire**, Dilbert is **dilbert@bigco.com**, and Grace Hopper is **ghop@cobol.gov**. All of them have the password 'secret' (What can I say, they are trusting souls!)

How to build from source

If you use Ant to build, begin by examining and modifying the **build.xml** file where all the build properties are defined at the top. Most of the Jar files that you will need are contained in the binary distribution, so you should be able to grab the miscellaneous files there and copy into the appropriate

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

directories in your source distribution. The build has a default target called **war-file** and a target called **clean**. Begin with the **clean** target. This will invoke the property verification targets that can detect some types of errors in your build configuration. Once you correct the errors and get the clean target working, you should be ready to build the War file. The War file that you build should look very similar to the War file in the binary distribution. The **clean** target must always be invoked manually. All of the other targets are incremental build steps.

Is the VLib example dependent on JPOX?

Most of the features of JDO that the VLib example uses are required features in every JDO 1.0 compatible implementation. In one case, the VLib example uses a JDO optional feature, nontransactional-read, which is implemented by JPOX and nearly every JDO implementation.

The VLib example also uses three vendor features in the JPOX implementation. Vendor features are features that are found in the vendor's implementation of JDO but are not specified by the JDO standard. When the same feature is found in another implementation, it generally has a different interface. I'll list the vendor features and say a few words about each.

First, VLib uses JPOX's ability to automatically generate, upon first encounter, the schema for data objects. This is a feature that is ready-made for an example application that will be deployed in many different development environments. Most JDO implementations have ways to create the schema automatically, but so far as I know, there are no plans to make automatic schema generation a standard feature of JDO.

Second, the VLib example uses the **toUpperCase** method in JDOQL (JDO's query language). This vendor feature is implemented in many JDO implementations and will be required in JDO 2.0.

Lastly, the VLib example uses JPOX's support of SQL to obtain a count of the query results for setting up the paging of search results. In an example program with a small test data set, it is obviously not necessary to spend much time worrying about how to obtain the count of query results, but I used this vendor feature for several reasons. One, the business logic to page results was already present in the original Virtual Library example. Two, paging results is a problem for many real-world applications. Three, many O/R mapping implementation of JDO support SQL queries. Finally, support for aggregates is part of the JDO 2.0.

In short, the VLib example as written will work only with JPOX, but the work to port it to any other JDO implementation is minimal. As this example evolves to use JDO 2.0, its dependency on JPOX will diminish.

What are the design choices in this example?

The VLib example demonstrates a Tapestry Web application using JDO as its model. In this sense, the example is a proof-of-concept. The integration of Tapestry and JDO can be done!

The example has other important goals as well. Since working example code is often used as a template for real-world applications, the VLib example strives to avoid bad code. There is no reason for you to learn bad lessons from an example. It also attempts to show good architectural design. In other words, not only does the example work, but it works in a good way. No doubt there are bugs in this code, and the bugs may go deeper than some code not working. If you spot bad code or bad design, speak up, and

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

help improve this example.

'Design patterns' is too strong a term for the designs decisions discussed here, since not all of these design decisions have been applied in multiple applications, and the classic exposition of a design pattern has not been adopted. Nevertheless, this example contains several design insights that are worth understanding. If you understand them, you can choose whether to apply them to the applications that you write.

Use one persistence manager per HTTP request

The first thing to understand when building a Web application using JDO, is to select and stick with the one-pm-per-request design pattern. This pattern was first described in the book **Using and Understanding Java Data Objects**. It is a simple concept, and it works without subtle errors.

The reason to obtain one persistence manager at the start of each request is the need to release the persistence manager at the end of the request. Persistence managers can consume limited resources which must be shared with other request handling threads. As a rule, they do not serialize, and therefore cannot be stored in the session.

The one-PM-per-request design pattern imposes real constraints on your application's design. We will encounter them at every turn throughout the rest of this discussion. The alternatives, however, can cause worse problems in the form of extra coding, increased complexity, the potential for subtle errors, and a failure to deliver the expected benefits. For further discussion of the alternatives, see Chapter 10, 'Using JDO in a Web Application' in **Using and Understanding Java Data Objects.**

In the one-pm-per-request design pattern, there is a static factory that produces persistence manager objects. In this example, the **PMLocator** class (contained within the **VLibDataService** source file) provides the factory by delegating to JDO's **PersistenceManagerFactory** class. Each time a request is serviced, a **VLibDataService** object is constructed, and during its initialization, it uses the locator to get a persistence manager. The important point is that the persistence manager factory is constructed only once for each Web container, but the persistence managers are obtained on each HTTP request. JDO is constructed to put most of the time into the construction of a persistence manager factory. The JDO implementation may use connection pooling, object pooling, and in some cases, data pooling, to reduce the computational burden of constantly getting and releasing persistence managers and the persistent objects they manage.

In the VLib example, the **VirtualLibraryEngine** class acts as the factory to produce the current request's **VLibDataService**. The **VirtualLibraryEngine** closes the **VLibDataService** (which thereby releases the persistence manager) in its **cleanupAfterRequest** method, which the Tapestry framework calls after every request. The VLib examples shows a clean and simple implementation of the one-pm-per-request pattern.

Encapsulate JDO transactions within the life-cycle of the service

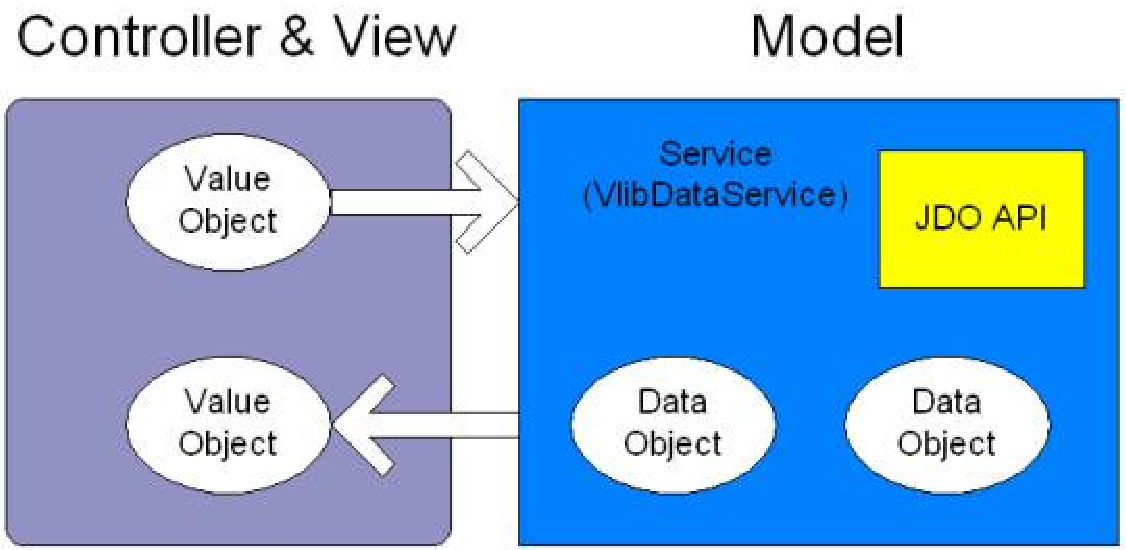
This is really just a corollary of the pattern above. Since JDO 1.0 transactions do not outlive the persistence manager, if you close the persistence manager you must also end the transaction. In JDO 2.0, extended optimistic transactions will be supported. For further discussion of extended optimistic transactions, see the section 'Add extended optimistic transactional semantics' towards the end of this tutorial.

Divide the model into service and data classes

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

One the one hand, data objects are plain-old-Java-objects (POJOs), and their classes can be simple JavaBeans. Data classes are identified in the JDO metadata. JDO fetches and stores the state of the managed objects of data classes without the need for your code to implement the magic. On the other hand, your code will need to invoke explicit JDO services to control transactions, to define and run queries, to iterate query results, to create new persistent objects, todelete existing persistent objects, and to create value objects.

Therefore, think of your model as a set of data classes wrapped by a service. What goes into and comes out of the service are value objects (more on this point in the next section). The service itself uses managed data objects. The following diagram illustrates the point.



In the VLib example, the data classes are **Book**, **Person**, and **Publisher**, and the application's service is the **VLibDataService**.

Use value objects in the controller and presentation code

JDO's management of persistent objects has its good points and its bad points. On the good side, a data object looks like a POJO. They are simple to design, and easy to code and understand. JDO handles all of the code required to shuffle state between the object and the database. On the bad side, JDO's management of the object can be a hindrance in some cases.

For example, JDO's management is a hindrance when changes to persistent objects need to be recorded in memory without having an active JDO transaction. JDO will not allow a managed object to be changed unless a transaction is active. (Surprisingly, JDO's nontransactional-write feature is not the solution it appears to be).

JDO's management is also a hindrance when the persistence manager is to be released, but the data object is still needed for later use. The persistent objects are no longer valid once the persistent manager has been released (in other words, its **close** method has been called). At the same time, it is natural in most Web applications to store some data objects in the session between requests. The Web container may serialize these objects between requests, and the persistence manager is not around to manage them.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

So it would appear that Web applications need data objects when they are not available. How is this problem solved? The answer is to create value objects which contain all of the information required for non-managed operations and for later updating of the corresponding managed object. Some people believe that separate classes should define the value objects, but in the classic words of Abe White, the architect of Kodo, 'data objects can be their own value objects.' By using the data classes to define both managed data objects and unmanaged value objects, both the code and its maintenance are simplified.

The entire process of creating a value object, modifying it, and later updating a managed data object with it is called 'detachment-attachment.' By creating value objects, the controller code can store and modify unmanaged data objects and sends them to the service to update managed data objects. JDO 2.0 will provide the detachment-attachment mechanism. In JDO 1.0, application code must do it. For the simple cases, it is not hard.

In the VLib example, the **DataObjectService** and the **VLibDataService** collaborate to insure that only value objects enter and leave the public methods of the **VLibDataService**.

For a simple example like VLib, the case can be made for using value objects only where needed and using managed data objects everywhere else. This would reduce the amount of garbage generated by servicing a request. Most real world Web applications are not simple. When confronting the inevitable complexity of real-world requirements, grand organizing principles that keep the architecture, design, coding, and debugging simple are generally worth more than a minor improvement in performance.

Encapsulate mutations of relationships in the service

Data classes like all Java classes define state and behavior. All of the state is defined in the member fields of the Java class. Some of this state is inherent to the data class. State stored in primitive fields and state stored in some system classes, such as **Integer**, **Date** and **String**, is the inherent state. It is inherent because the state applies directly to the object. Other state defines the relationships between data objects. A **Person** may relate to a **Book** which the person has borrowed. An **Invoice** may contain **LineItems**.

The attachment code to handled modified relationships is more complicated than the attachment code to handle modified inherent attributes. Since in JDO 1.0 the application's code must perform the attachment, it is convenient to delegate to the data service all operations that modify the relationships between data objects. The support for attachment in JDO 2.0 will give us a reason to reevaluate this argument.

When writing a Web application, there is another reason to encapsulate mutations of relationships in the service. Often the relationships between data objects are managed by the user through the user interface. In HTML, the objects being referred to by various controls can be uniquely identified by the identity string of the persistent object. When the user borrows a book, the natural thing to expect from the HTML control is the identity string of the new owner. Since the controller code does not know which person goes with the identity string, it is natural to delegate the task to the data service. This reason will remain even with the advent of JDO 2.0.

In the **VLibDataService**, there are several methods that modify the relationships between data objects, such as **borrowBook**, **returnBook**, and **transferBooks**. Each of these service methods look up persistent **Book** objects and persistent **Person** objects and call the book's **setHolder** and **setOwner** methods.

Define an interface for your value object's operations.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

It is often the case that a value object should not change some state that a managed object can and must change. In the discussion above, the **setHolder** and **setOwner** methods of the **Book** class are not operations for value objects. Combine this with the question of what is the best value object implementation, and you have the perfect prescription for an interface. By making all of your controller and presentation code use an interface to the value objects, you create the means to control what operations the controller and presentation code perform and what implementation will perform it.

In the case of VLib example, the **Book**, **Person**, and **Publisher** data classes each have one of the value interfaces, **IBook**, **IPerson**, and **IPublisher**.

Encapsulate transactions within the service when possible

JDO does not, nor should it, encapsulate transactions. Where transactions are bounded and which code controls the boundaries is a design decision that varies from application to application. In the case of Web applications, since the life-time of the service and the persistence manager it uses is congruent with the processing of the HTTP request (always assuming that you have adopted the one-pm-per-request design pattern), transactions that JDO 1.0 can enforce must be short lived.

Since the transaction is already confined to the duration of the HTTP request processing, it makes sense to go further and confine it to the duration of a data service method call. While not necessary, this confinement simplifies the controller's code. If you find as I do that the most precious development resource is developer time, it pays to adopt the KISS solution as the first solution.

In the case of the VLib application, all JDO transactions live within the confines of a **VLibDataService** method call.

Add extended optimistic transactional semantics

Concurrent access to the same data is a real problem in database Web applications. In the VLib example, two people can be viewing the same book, and independently decide to borrow it. Without concurrency control, the last borrower wins. Since the view and the request to borrow it are two separate HTTP requests, JDO transactions do not apply (always assuming that you are using the one-pm-per-request design pattern.).

What is required can be called 'extended optimistic transactions' (which I suppose we could shorten to EOTS or 'e-oughts'). With EOTS, the code executing user A's request to borrow the book detects that the book has been changed since the book has been viewed. It is possible for the application to implement EOTS using JDO 1.0. There is an example of an EOTS implementation in the book **Using and Understanding Java Data Objects.**

The VLib example does not implement extended optimistic transactional semantics for two reasons. One, this was and remains a problem in the original Virtual Library example. Two, JDO 2.0 will provide EOTS for detached objects and it will also provide the tools to write a trivial implementation of EOTS even when the detached objects are discarded.

So this defect remains in the VLib example and is expected to be corrected with the evolution of the example as JDO 2.0 becomes available.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.13 Tapestry : Petstore

JPOX and Tapestry : The Developer's Guide to the Petshop Example

Document version 0.91.
© 2004 David Ezzio. All rights reserved.

The Petshop example uses Tapestry and DataNucleus. This document is a guide for building the Petshop example from source. It also discusses the recommended design choices followed by the example code, and it compares the use of DataNucleus to the use of the Data-Access-Object (DAO) design pattern. DataNucleus provides a downloadable sample with the code described here.

What is the Petshop example?

This example, called Petshop, is a port of the Petshop example (Version 0.41) created and distributed by Luis Neves. Like the VLib example that is also distributed with DataNucleus, the Petshop example shows how a Tapestry Web application can use JDO.

The original Petshop example is, I believe, a recreation of the J2EE Petstore example. The fictitious business case for the Petshop is an on-line business that sells pets. The Petshop Web application allows people to browse or search the catalog of pets, select pets for purchase, and complete the order. Luis Neves shows off Tapestry#s support for internationalization by supporting both a Portuguese and English version of the site. This port continues to do the same.

The original Petshop example uses JDBC and the Data Access Object (DAO) pattern to access the database and provide the persistence layer. The ported example uses JDO to provide the persistence layer.

Who is most likely to benefit from this example?

Developers who are familiar with or learning Tapestry will benefit from this example. Developers who are very familiar with the DAO pattern using JDBC and who are thinking of adopting JDO will benefit from comparing this version of the example with the original.

Technical managers who are evaluating whether to use JDO in their development process will benefit from considering the metrics drawn from this example. (See last section.)

What is Tapestry?

Tapestry is a Web application framework from Apache. It can be found at http://jakarta.apache.org/tapestry.

Web applications within Tapestry follow the model-view-controller (MVC) pattern. The view consists of HTML templates, and specifications in XML for JavaScript, as well as style sheets and graphic resources.

The controller consists of Java code with page and component wiring in XML. Tapestry does not provide a model, making it an ideal partner for JDO.

Tapestry abstracts from the multi-threadedness of the Servlet API and provides tight coupling between the HTTP request and response handling code. The same components that participate in the rendering of page also accept the user#s response to the page. The handshaking between HTTP responses and the Java code is handled by the framework, so that most dynamic information presented on the page and most user input to the page are represented as properties of the Tapestry page or its components. The user#s next request action results in an invocation of a listener method on the page or one of its components. Every Tapestry page and component is single-threaded by design, and contrary to what you might be inclined to believe, it performs quite well because of object pooling.

How does the Petshop example differ from the original?

The Petshop example is a port (or fork) of the Petshop example that is distributed by Luis Neves.

Unlike the original which uses the DAO pattern and JDBC to implement the model, the Petshop example distributed by DataNucleus uses JDO.

In general, most of the presentation files are unchanged, and those that are changed, are changed very little. The controller code varies class-by-class from no change, to one or two little changes, to many small changes. The model code has been extensively modified, although there is a family resemblance to the original code.

How do I set up the Petshop example in my development environment?

This example is distributed as source only. This section documents the steps you need to follow to successfully build the example from source.

The preliminaries

For many of you, the preliminary configuration steps for a development environment are no-ops, since you will have several of these pieces already installed and ready to use. For the rest of you, this checklist will help you pick up the pieces that you need.

- Install JDK 1.4 or later. This Petshop example uses the version of **java.lang.Throwable** with a nested cause exception that was introduced in JDK 1.4. So far as I know, this is the only dependency on JDK 1.4.
- Install Ant 1.5.3 or later. This product is available from the Apache Foundation.
- Install Tapestry 3.0. This product is available from the Apache Foundation. If you are just learning Tapestry, you should get the Hangman example working. The book **Tapestry in Action** from Manning Publications can be quite useful.
- Tapestry requires two libraries that are not distributed with it. Obtain the JavaAssist library, version 2.6 or later, from the JBoss Group. Obtain the Object Graph Navigation Language library, version 2.6.3 or later from OGNL Technology Inc.. If you create a directory named **extras** under your Tapestry home directory and place these two jar files there, you#ll have one less configuration step in the build file.
- Install DataNucleus 1.1.0.A2 (or later). If you are new to JDO, you should work through the

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

DataNucleus tutorial. The book **Using and Understanding Java Data Objects** from Apress can be quite useful for understanding both the details of JDO and the big picture.

The DataNucleus 1.1x implementation is a preview implementation of the coming JDO 2.0 specification. This example uses some mapping features found in this branch of DataNucleus that are not found in the 1.0 branch, which is a JDO 1.0.1 implementation. Other than the new mapping features, the Petshop example does not use any of the new features found in JDO 2.0. A later section of this tutorial discusses the reason for using the mapping features.

- Install the McKoi database version 1.0.2 or later, a lightweight, Java only SQL database. This open-source database is available from http://www.mckoi.org.

  A ready to use set of data files is distributed with this example in the **mckoi-db** directory as a zip file. Unzip **PetshopDB.zip** to the root directory of your McKoi install.

  The command files **run.bat** and **stop.bat** are included for Windows users. These start and stop the database server. If you are running on Linux or some other environment, you#ll have to create the scripts to start and stop the database.

  In addition, there is a **console.bat** command file to startup the interactive console to the McKoi Petshop database. This tool can run only when the database server is stopped. Again, those who are not using Windows will have to create the script to start the console. Use this tool to examine the data in the database.
- Since the Petshop example is a Web application, you need a J2EE Web container, such as Tomcat from the Apache Foundation.
- Copy the McKoi jbdc driver library **mkjdbc.jar** found in the McKoi home directory to the appropriate location for your Web container. For Tomcat, the appropriate location is **<tomcat-home>/common/lib**.
- Unzip the Petshop example code to a directory of your choice.

Configure the build file

Before running the build file you must configure it for your development environment. To do this, assign the correct values for your development environment to the properties defined near the top of the build file. Here is a short description of each property found there.

- **webapps.dir** : Path to the directory where the Web application war file should be deployed.
- **jpox.home** : Path to the DataNucleus home directory.
- **jpox-jar** : Path to the DataNucleus runtime jar file.
- **jpox-enhancer-jar** : Path to the DataNucleus enhancer jar file.
- **bcel-jar** : Path to the BCEL jar used by the DataNucleus enhancer.
- **tapestry.home** : Path to the Tapestry home directory.
- **tapestry-extras.dir** : Path to the directory where you placed the extra libraries required by Tapestry. (See above section.)

Build and deploy

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The two main build targets are **clean** and **deploy**. Invoke Ant in the customary way. The Ant download provides ample documentation for getting started with Ant.

Invoke first the **clean** task, then the **deploy** task. In both cases, you should see a message that indicates that the build was successful.

Test the Petshop Web application

After the task deploys the Petshop Web application, start up the Web server if it is not already running. Visit the site by using an URL like the following, **http://localhost:8080/petshop/app** . The host name and port may differ for your development environment. If everything has gone well, you#ll see a web page like the following.

The site is easy to use. It has a link to the help page that tells you more. The password for the #j2ee# user is #j2ee#.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

What databases does the Petshop example support?

The original Petshop example comes with data definition language (DDL) scripts to initialize a McKoi database, a Microsoft SQL Server database, or a PostgreSQL database. DataNucleus itself supports a large number of relational databases.

When building applications that use JDO, real world requirements often include the need to support an existing database schema. For that reason, I adopted the requirement that the port of Petshop to DataNucleus would support the existing McKoi data schema. This requirement led to three results. One, the port does support the same McKoi data schema as the original Petshop example. No changes whatsoever to the existing tables. DataNucleus does add its own DataNucleus_TABLE, which it uses to keep track of the persistent classes. Two, in order to get the mapping functionality that I needed to meet this requirement, I had to adopt the DataNucleus 1.1 line which has already implemented some of the mapping functionality found in the coming JDO 2.0 specification. Three, to minimize the amount of testing, I limited myself to only the McKoi database schema.

If someone wants to support either Microsoft SQL Server or PostgreSQL using the original DDL (shipped with this example), the changes required, which may not be many, should be limited to the JDO metadata. If someone wants to lift the restriction that an existing schema must be supported, again the metadata must change, and in addition, a means must be devised to initialize the test data for the schema that DataNucleus will automatically generate.

Is the Petshop example dependent on DataNucleus?

Most of the features of JDO that the Petshop example uses are required features in every JDO 1.0 compatible implementation. In one case, the Petshop example uses a JDO optional feature, nontransactional-read, which is implemented by DataNucleus and nearly every JDO implementation.

The coming JDO 2.0 standard will likely include better support for specifying the object-to-relational mapping in the JDO metadata. The DataNucleus 1.1.x line of releases is currently a preview version for the coming JDO 2.0 standard as expressed in the early release draft. For the moment, the JDO metadata for the Petshop is definitely tied to the DataNucleus 1.1.x line, but it is anticipated that this metadata will become standard JDO 2.0 metadata with little change. As it stands now, the Petshop metadata file does not use any extension tags, which are the tags that allow the use of vendor specific features in the metadata.

The Petshop example also uses the expected JDO 2.0 support for the **toLowerCase** method in JDOQL (JDO#s query language).

In short, the VLib example as written will work only with DataNucleus 1.1.x. It will evolve, as the JDO 2.0 standard and as work on DataNucleus 1.1.x continues, to become dependent only on JDO 2.0.

Does the Petshop example follow the design choices recommended in the tutorial for the VLib example?

In the tutorial for the VLib example, another example of a Tapestry Web application that uses DataNucleus, I identified several design choices and described why I thought they were good choices. Did I follow those recommendations in this example? The following checklist provides the rundown.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

- *One persistence manager per HTTP request*

  Yes. I really think this rule is hard and fast. You won't catch me breaking it anytime soon.
- *Encapsulate the transactions within the lifetime of the application service that calls JDO*

  Yes. In JDO 1.0 this rule is a corollary of the first rule above.

  In JDO 2.0, the Web application may break this rule by using the support for extended optimistic transactions (EOTS) that comes with the detach/attach API. The Petshop example does not have a great need for EOTS, since by its nature it is a mostly-read and write-and-forget application.
- *Divide the model into service and data classes*

  Yes. Also added shopping cart classes to the model in a separate package. In a real world application these would likely also be persistent classes, but in this example, the cart exists only in the servlet session space.
- *Use values objects in the controller and presentation code*

  Yes. This design choice is excellent for separating the areas of concern. As in the VLib example, I use unmanaged data objects for the value objects. This example does not yet use the detach/attach API. It is still using the homegrown make-a-clone value object pattern that was used in VLib. This pattern works with JDO 1.0. As JDO 2.0 becomes more defined, the example will evolve to use the detach/attach API.
- *Encapsulate mutations of relations in the service methods*

  Not applicable. This application does not change any relationships between the persistent objects. It does write out orders and line items, but its job in this regard is to write and forget. If the shopping cart were persistent, then persistent relationships would be modified as items are added and removed from the cart.
- *Define an interface for the value objects*

  Yes. The original Petshop example already had these. Their use was retained.
- *Encapsulate JDO transactions within the service methods when possible*

  Yes. Another code simplifying design choice.
- *Use extended optimistic transactions (EOTS) to guarantee that objects when changed are consistent with their viewed state.*

  Not applicable. This is a read-mostly application. The writes are write-and-forget.

Good news for management

My primary motivation in doing this port was to demonstrate the advantages of using JDO when compared directly to coding the DAO pattern with the use of JDBC.

The Data Access Object (DAO) design pattern provides an adapter between plain, old, Java objects (POJOs) and various databases that can be accessed using JDBC. In essence, the DAO pattern requires that you write the persistence layer for your data classes, and it gives you guidance on how to go about it. By writing DAO code, you expend business resources to create the infrastructure for your application. Expending development resources on coding the DAO pattern is common and not inconsequential.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

The following table shows the impact on the Petshop example of switching from the code that implements the DAO pattern with JDBC to the code that uses JDO. Table 1-1 compares original DAO version of Petshop to the new JDO version of Petshop based on two metrics. The SLOC metric is the count of the source lines of code. This count does not include blank lines or comment lines. The CC metric is the count of classes, including interfaces. The Petshop application is broken down into two sections, the model and the controller. Since the view is HTML templates and XML files, it is not included in the analysis of Java code. The view was nearly unchanged by the port.

Table 1-1: Metrics comparing the use of DAO to JDO

	SLOC	*CC*	*SLOC*	*CC*	*SLOC*	*CC*
Model	2773	60	1646	34	-41%	-43%
Controller	2156	36	2156	36	0	0
Both	4929	96	3802	70	-23%	-27%

As the table shows, the port had no impact on the controller code by these metrics. On the other hand, the impact on the model is dramatic. The amount of model code is reduced by over forty percent. Less code written means less time creating it, less time debugging it, less time testing it, and less time maintaining it. Overall, the amount of Java code for the entire application is reduced by approximately twenty-five percent. Most managers will find it hard to ignore numbers like these.

If you believe as I do, that it is easier to learn to use JDO than to learn to code the DAO pattern with JDBC, and if you believe, as I do, that using a JDO implementation will lead to a more robust and a better performing persistence layer than having your best developers create it from scratch, then the reduction in code coupled with these advantages creates a compelling argument for using JDO in your next development project.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

20.14 Web sample (LongPlay)

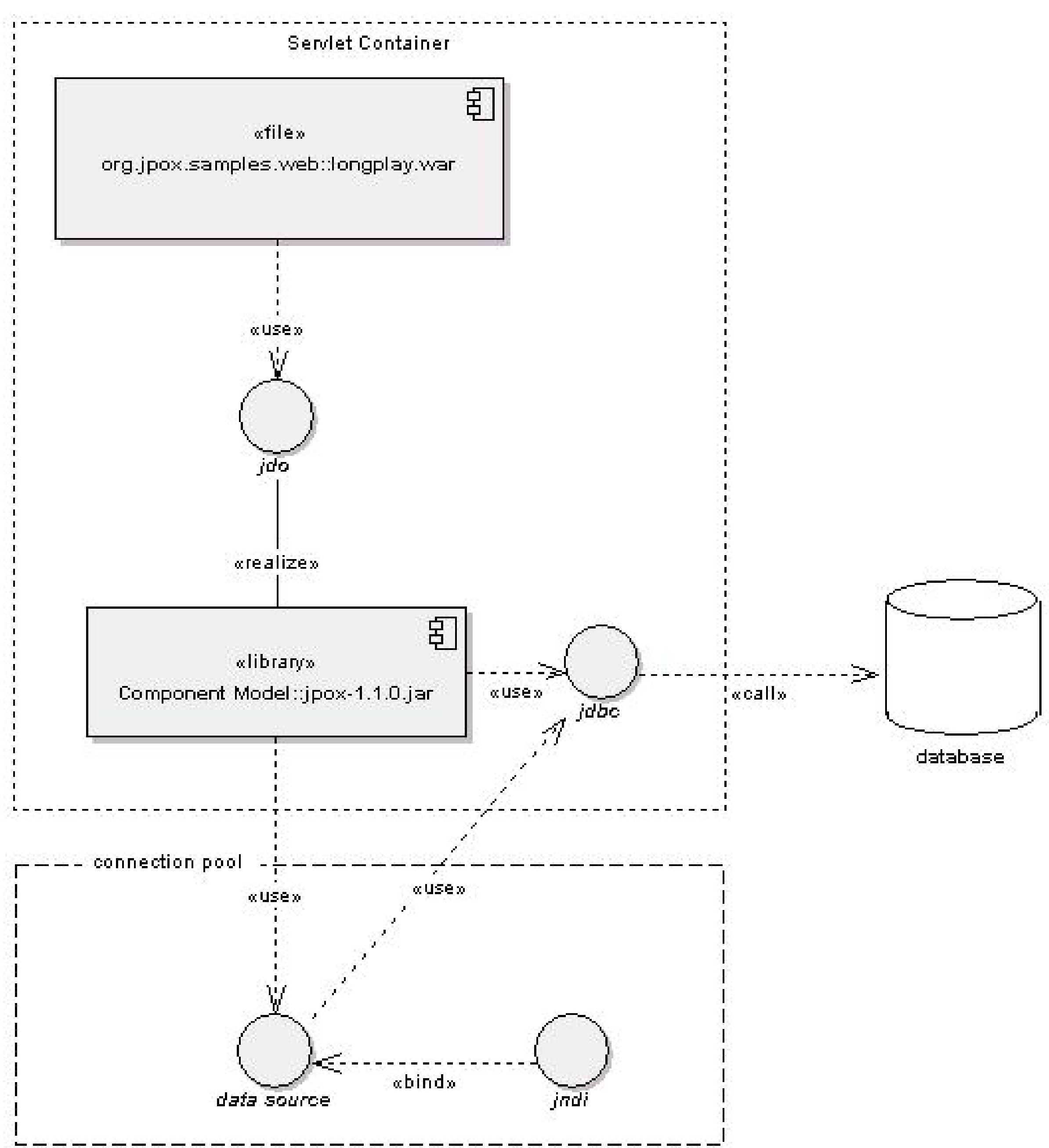
JPOX - Longplay Web Sample application

Longplay is a virtual company that sells video and music products. Longplay does not have a web store seller site, but a list of the commercialized products on web.

Architecture

The Longplay web site runs inside a servlet container and depends on JDO for persistence services. The Longplay web site uses a database connection pool to improve use of resources.

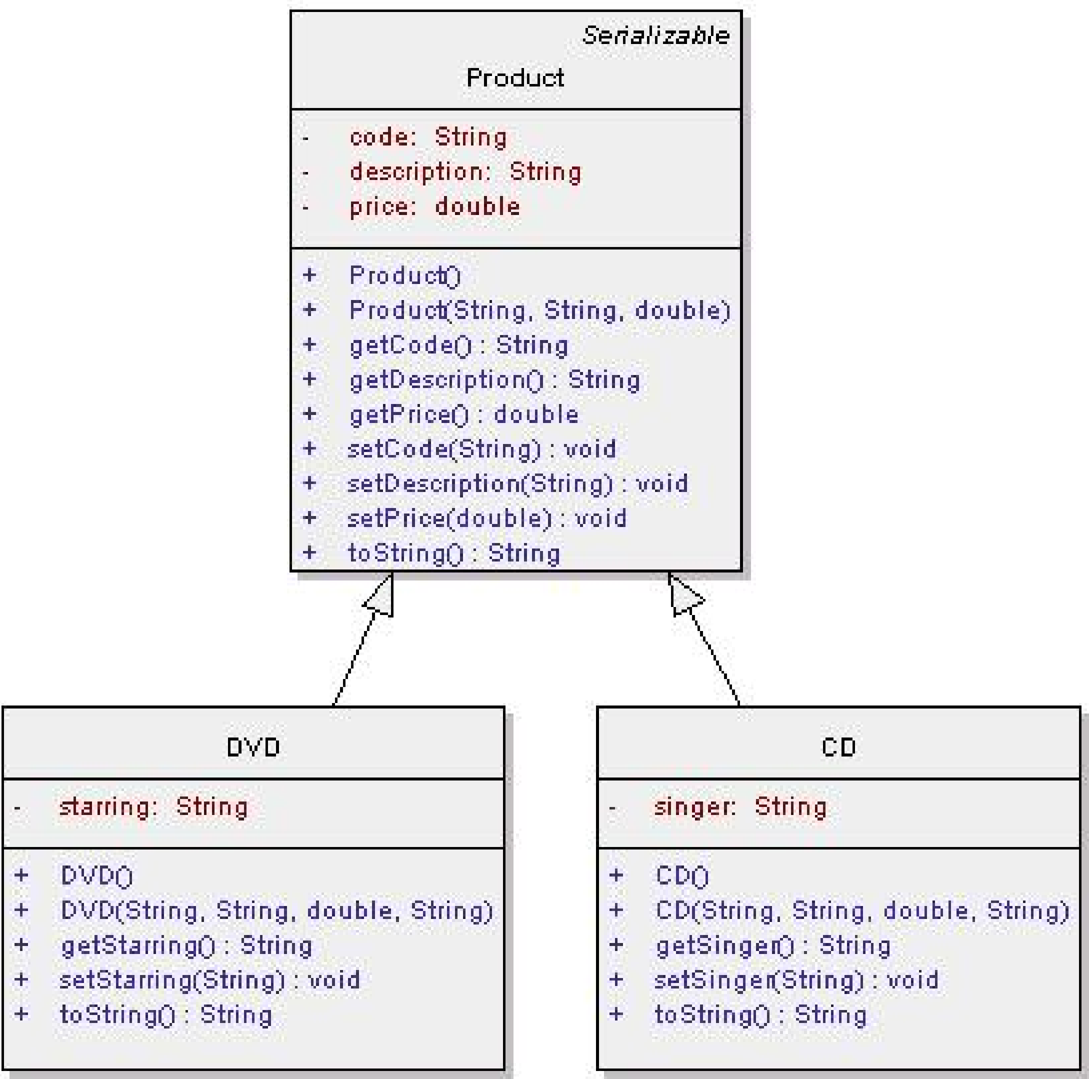
©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Domain model

The Longplay web site development process is centered and driven by the domain model. The domain model is composed of three classes: Product, DVD and CD. These three classes are persistent and its instances are going to be store in the database.

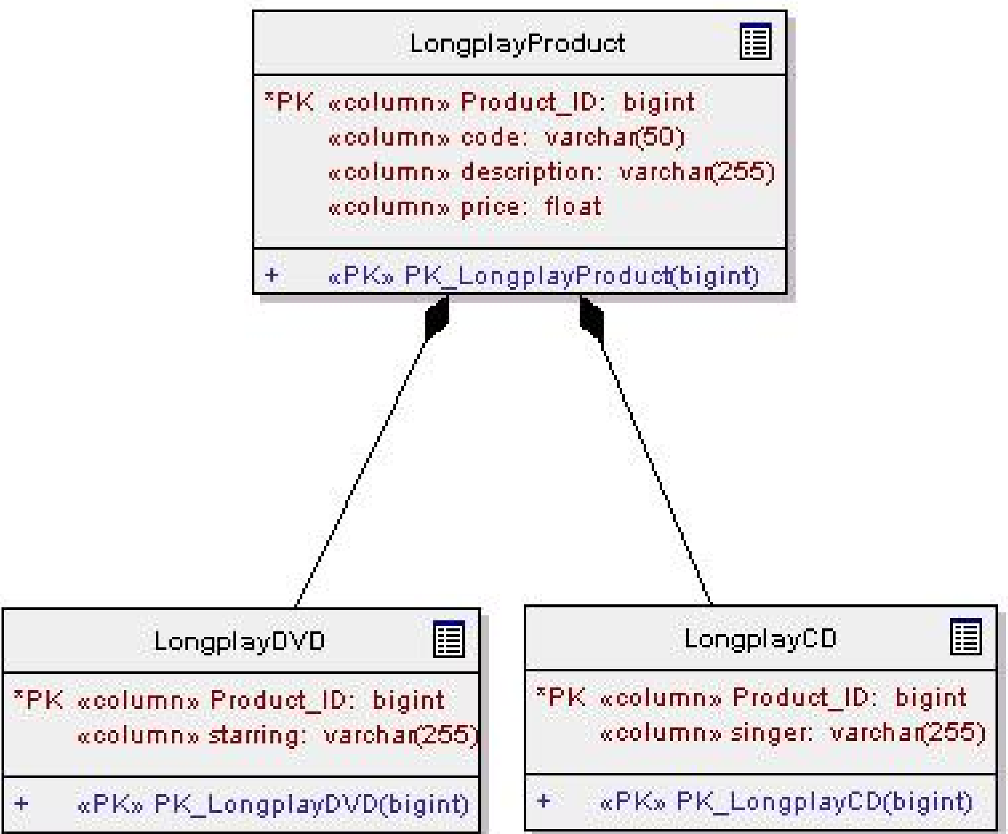
©2008-2009, DataNucleus • ALL RIGHTS RESERVED



Database model

You've already heard the saying: "Behind every great man is a great woman", but maybe you've not heard yet the analogous version to applications and databases: "Behind every great application is a great database model". Data is the sustaining factor in typical business applications and put some effort in the database model is far than important to allow consistent storage and efficient access to data.

You can start your database model from scratch or from an existing schema, JPOX support both modes. Therefore creating a new schema or using an existing one will have the same effect in the end. It's important to note that you will have to make the correspondences in a JDO file to match the models.



Domain and Database model matching

The domain and database models do not have to follow one java class to one database table, and you are allowed to normalize or denormalize both domain models and database models.

To make this happen transparently, you have to map the domain model to the database model or vice versa. In JDO, you create JDO metadata files for this purpose.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="org.jpox.samples.web.model">
        <class name="Product" table="LongplayProduct">
            <!-- a primary-key in the table that does not have a correspondent field
-->
            <datastore-identity strategy="identity" column="Product_ID"/>

            <field name="code">
                <column name="code" jdbc-type="VARCHAR" length="50"/>
            </field>
            <field name="description">
                <column name="description" jdbc-type="VARCHAR" length="255"/>
            </field>
            <field name="price"/>
                <column name="price"/> <!-- let DN select the best sql type -->
            </field>
        </class>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        <class name="DVD"
persistence-capable-superclass="org.jpox.samples.web.model.Product"
table="LongplayDVD">
            <inheritance>
                <join>
                    <column name="Product_ID"/>
                </join>
            </inheritance>
            <field name="starring">
                <column name="description" jdbc-type="VARCHAR" length="255"/>
            </field>
        </class>
        <class name="CD"
persistence-capable-superclass="org.jpox.samples.web.model.Product"
table="LongplayCD">
            <inheritance>
                <join>
                    <column name="Product_ID"/>
                </join>
            </inheritance>
            <field name="singer">
                <column name="description" jdbc-type="VARCHAR" length="255"/>
            </field>
        </class>
    </package>
</jdo>
```

The previous snippet shows the JDO metadata for our sample, with the domain model mapped to our database model and several fine tunings (jdbc-type, length, colum and table names). We will let JPOX create the schema for us in the database.

Pitch on the proper solution

To adjust for maximum performance and development experience, we select a set of features to apply in our application:

- **Schema created by JPOX**
  Configured at the PersistenceManagerFactory creation by a property, *org.jpox.autoCreateSchema*, defined in the jdo.properties file. For further reference, see the PMF Guide.
- **Using a DataSource available inside an Application Server or Web Server**
  Configured at the PersistenceManagerFactory creation by a property, *javax.jdo.option.ConnectionFactoryName*, defined in the jdo.properties file.

  ```
  javax.jdo.option.ConnectionFactoryName=jdbc/ds
  ```

  For further reference, see Connection pooling.
- **Custom table and column naming**
  User customized table and column names are used in this example.

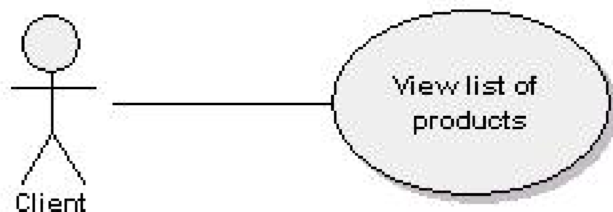
  ```
  <?xml version="1.0"?>
  ```

```
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="org.jpox.samples.web.model">
        <class name="Product" table="LongplayProduct">
            <field name="description">
                <column name="description" jdbc-type="VARCHAR" length="255"/>
            </field>
        </class>
    </package>
</jdo>
```

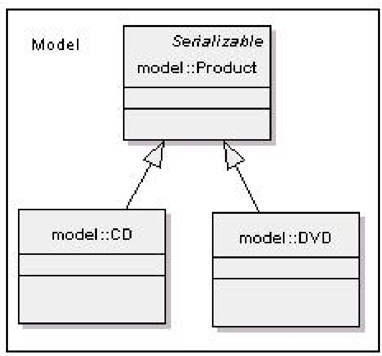
For further reference, see Meta-data.

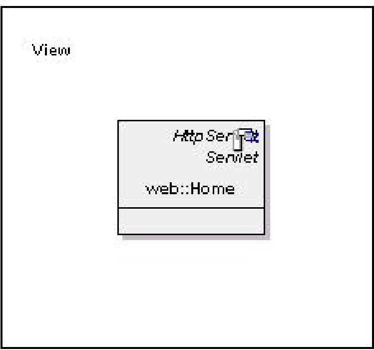
Implementing the use cases

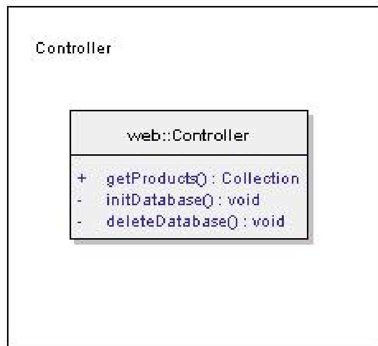
The JDO stuff is done. We are ready to implement the use cases. Our application is very simple, and all we have to do is implement a use case to list the Longplay's products.



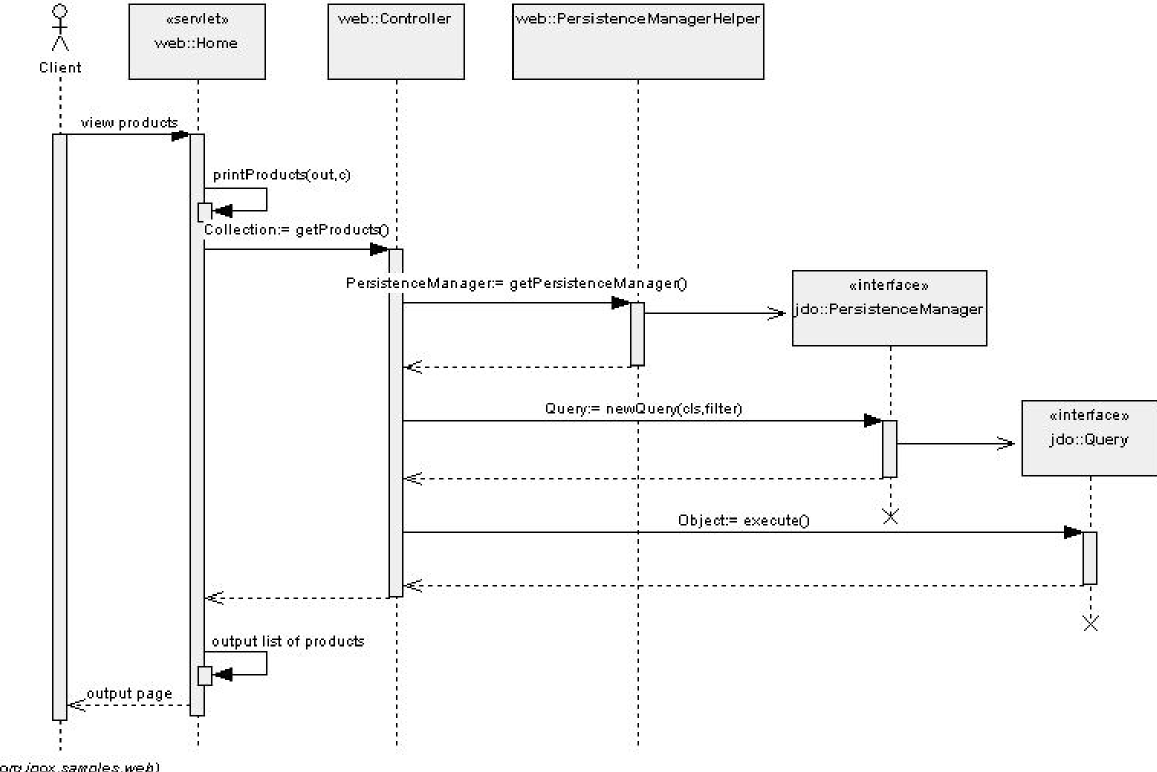
To implement the View list of products Use Case, we use a simple Model-View-Controller pattern. The sequence of actions described by the Use Case is implemented by the Controller and the observable value to Actors are presented in Views, in our use case we have only one View.







©2008-2009, DataNucleus • ALL RIGHTS RESERVED



The above diagram display the objects in collaboration to realize our Use Case. Our use case implementation is just a few lines of code. It gets a PersistenceManager instance, opens a transaction, fetch the contents from the database and make the results transient to be used outside a PersistenceManager context.

```
    /**
     * Returns a collection of Products
     * @return A collection of Product
     */
    public Collection getProducts()
    {
        PersistenceManager pm = PersistenceManagerHelper.getPersistenceManager();
        try
        {
            pm.currentTransaction().begin();
            Collection c = (Collection)
pm.newQuery(pm.getExtent(Product.class,true)).execute();
            pm.retrieveAll(c);
            pm.makeTransientAll(c);
            Collection products = new ArrayList(c);
            pm.currentTransaction().commit();
            return products;
        }
        finally
        {
            if (pm.currentTransaction().isActive())
            {
                pm.currentTransaction().rollback();
            }
            pm.close();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }
    }
```

Running

The Longplay application is organized with the following structure:

- **src/java** : source files
- **src/webapp** : html, jsp, gif and jpg files
- **src/webapp/WEB-INF** : compiled classes, jdo files and the web.xml file
- **lib** : The placeholder for jar files. **Don't forget to put the jar files required by JPOX to run**
- **build.xml** : Ant script for building using Ant
- **project.xml, project.properties, maven.xml** : Maven build files.
- **jpox.properties** : File defining the data source location and PMF controls

You can build the LongPlay application using either Maven or Ant, and it is available to download from SourceForge.

Summary

This tutorial presented the overall steps required to develop a web application using JDO. We hope that you are now ready to develop web applications using JPOX.

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

21.1 The JPA Tutorial

DataNucleus - Tutorial for JPA

Background

An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. Step 1 : Design your domain/model classes as you would do normally
2. Step 2 : Define their persistence definition using Meta-Data or annotations.
3. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
4. Step 4 : Generate the database tables where your classes are to be persisted.
5. Step 5 : Write your code to persist your objects within the DAO layer.
6. Step 6 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-tutorial-*").

Step 1 : Create your domain/model classes

Do this as you would normally. JPA places some constraints on persistable classes.

- Class can't be final
- Class can't have final methods
- Class must have a non-private default constructor.
- Class must have a field that stores the "identity"

To give a working example, let us consider an application handling products in a store.

```
package org.datanucleus.samples.jpa.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    protected Product()
    {
    }

    public Product(String name, String desc, double price)
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}
```

```
package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
```

So we have inheritance between 2 classes. Some data in the store will be of type *Product*, and some will be *Book*. This allows us to extend our store further in the future and provide *DVD* items for example, and so on. In traditional persistence using, for example EJB CMP, this cannot be persisted directly. Instead the developer would have to generate a level above the entity beans to define the inheritance. This is messy. With JPA, we don't have this problem - we can simply throw objects across to JPA and it will persist them, and allow them to be retrieved maintaining their inheritances.

Step 2 : Define the Persistence for your classes

You now need to define how the classes should be persisted, in terms of which fields are persisted etc. With JPA you can define what classes are persistable with either MetaData (XML) or Java5 annotations. In this example we will use a mixture of both to highlight best practice. Let's start with which classes/fields are to be persisted. Here we use annotations. For example, for our 2 domain classes

```
package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product
{
    @Id
    long id;

    @Basic
    String name = null;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    @Basic
    String description = null;

    @Basic
    double price = 0.0;

    ...
}
```

```
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product
{
    @Basic
    String author=null;

    @Basic
    String isbn=null;

    @Basic
    String publisher=null;

    ...
}
```

So we have annotated both classes as "Entity" meaning that they are persistable. We have annotated the fields to be persisted, and the "id" field as being to store the identity. In addition we have defined that the classes will have one table in the datastore for each class. Now lets get on to the details of how they should be represented in an RDBMS datastore. We are going to define this in a MetaData file since it is better to keep schema information separate from persistence information. So we define a file *orm.xml*

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">
    <description>DataNucleus JPA tutorial</description>
    <package>org.datanucleus.samples.jpa.tutorial</package>
    <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
        <table name="JPA_PRODUCTS"/>
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="name">
                <column name="PRODUCT_NAME" length="100"/>
            </basic>
            <basic name="description">
                <column length="255"/>
            </basic>
        </attributes>
    </entity>
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
    <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
        <table name="JPA_BOOKS"/>
        <attributes>
            <basic name="isbn">
                <column name="ISBN" length="20"></column>
            </basic>
            <basic name="author">
                <column name="AUTHOR" length="40"/>
            </basic>
            <basic name="publisher">
                <column name="PUBLISHER" length="40"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

We now need to tell JPA which classes we are going to persist. We do this via a file *persistence.xml* stored under "META-INF/" at the root of our package structure.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <mapping-file>org/datanucleus/samples/jpa/tutorial/orm.xml</mapping-file>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
    </persistence-unit>

</persistence>
```

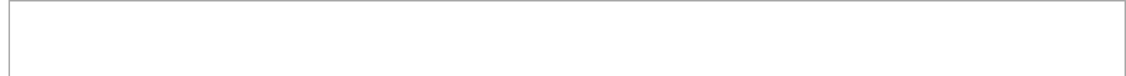
So we have defined a *persistence-unit* called "Tutorial". This packages our classes to be persisted and we use that name later.

Step 3 : Instrument/Enhance your classes

DataNucleus relies on the classes that you want to persist being PersistenceCapable. That is, they need to implement this Java interface. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable.

DataNucleus provides a byte-code enhancer for instrumenting/enhancing your classes. You will need to obtain the DataNucleus Enhancer JAR to use this.

To understand on how to invoke the enhancer you need to visualise where the various source and metadata files are stored



©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
src/java/META-INF/persistence.xml
src/java/org/datanucleus/samples/jpa/tutorial/orm.xml
src/java/org/datanucleus/samples/jpa/tutorial/Book.java
src/java/org/datanucleus/samples/jpa/tutorial/Product.java

target/classes/META-INF/persistence.xml
target/classes/org/datanucleus/samples/jpa/tutorial/orm.xml
target/classes/org/datanucleus/samples/jpa/tutorial/Book.class
target/classes/org/datanucleus/samples/jpa/tutorial/Product.class

lib/persistence-api.jar
lib/jdo2-api.jar
lib/datanucleus-core.jar
lib/datanucleus-enhancer.jar
lib/datanucleus-rdbms.jar
lib/datanucleus-jpa.jar
lib/asm.jar
```

So you see that we have *persistence-api.jar* which is "JPA", *jdo2-api.jar* which is "JDO" (and DataNucleuss JPA support is built on top of JDO), *datanucleus-jpa.jar* which contains DataNucleuss support for JPA and Java5, and in addition dependent jars.

The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven project to do this for you.

```
Using Ant :
ant compile


Using Maven :
maven java:compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

```
Using Ant :
ant enhance

Using Maven : (this is usually done automatically after the java:compile goal)
maven datanucleus:enhance


Manually on Linux/Unix :
java -cp
target/classes:lib/datanucleus-enhancer.jar:lib/datanucleus-jpa.jar:lib/jdo2-api.jar:
         lib/persistence-api.jar:lib/datanucleus-core.jar:lib/asm.jar
     org.datanucleus.enhancer.DataNucleusEnhancer
     -api JPA
     -persistenceUnit Tutorial

Manually on Windows :
java -cp
target\classes;lib\datanucleus-enhancer.jar;lib\datanucleus-jpa.jar;lib\jdo2-api.jar;
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        lib\persistence-api.jar;lib\datanucleus-core.jar;lib\asm.jar
    org.datanucleus.enhancer.DataNucleusEnhancer
    -api JPA
    -persistenceUnit Tutorial

[Command shown on many lines to aid reading - should be on single line]
```

This command enhances the .class files that are defined by the persistence.xml file. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a *ClassNotPersistenceCapableException* thrown.

You can alternatively build your application using either Maven or Ant instead of the above manual method. The use of the enhancer with these 2 build systems are documented in the Enhancer Guide

The output of this step are a set of class files that represent PersistenceCapable classes.

Step 4 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the *src/java/META-INF/persistence.xml* file with your database details. Here we have a sample file (for HSQLDB) that contains

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <mapping-file>org/datanucleus/samples/jpa/tutorial/orm.xml</mapping-file>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <properties>
            <property name="datanucleus.ConnectionDriverName"
value="org.hsqldb.jdbcDriver"/>
            <property name="datanucleus.ConnectionURL"
value="jdbc:hsqldb:mem:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>

</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Using Ant :
ant createschema


Using Maven :
maven datanucleus:schema-create


Manually on Linux/Unix :
java -cp
target/classes:lib/jdo2-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
         lib/datanucleus-rdbms.jar:lib/datanucleus-jpa.jar:lib/{jdbc_driver.jar}
     org.datanucleus.store.rdbms.SchemaTool
     -create
     -api JPA
     -persistenceUnit Tutorial

Manually on Windows :
java -cp
target\classes;lib\jdo2-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
         lib\datanucleus-rdbms.jar;lib\datanucleus-jpa.jar;lib\{jdbc_driver.jar}
     org.datanucleus.store.rdbms.SchemaTool
     -create
     -api JPA
     -persistenceUnit Tutorial

[Command shown on many lines to aid reading. Should be on single line]
```

This will generate the required tables, indexes, and foreign keys for the classes defined in the annotations and *orm.xml* Meta-Data file.

Step 5 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an *EntityManagerFactory* for our "persistence-unit" called "Tutorial" (defined in persistence.xml earlier).

Now that the application has an EntityManager it can persist objects. This is performed as follows

```
Transaction tx = em.getTransaction();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
try
{
    tx.begin();

    Product product = new Product("Sony Discman", "A standard discman from Sony",
49.99);
    em.persist(product);

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    em.close();
}
```

Please note that the *finally* step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.

```
Transaction tx = em.getTransaction();
try
{
    tx.begin();

    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();

        ... (use the retrieved object)
    }

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    em.close();
}
```

If you want to delete an object from persistence, you would perform an operation something like

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Transaction tx = em.getTransaction();
try
{
    tx.begin();

    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    em.close();
}
```

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JPA book will provide many examples.

Step 6 : Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- Any JDBC driver classes needed for accessing your datastore
- The JDO2 API JAR (defining the JDO2 interface) - DataNucleus JPA requires this currently
- The JPA1 API JAR (defining the JPA1 interface)
- The **DataNucleus Core** JAR

After that it is simply a question of starting your application and all should be taken care of. You can access the **DataNucleus JDO** Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

```
Using Ant (you need the included datanucleus.properties to specify your database)
ant run


Using Maven ("runtutorial" goal included in the download JAR):
maven runtutorial
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
Manually on Linux/Unix :
java -cp
lib/persistence-api.jar:lib/jdo2-api.jar:lib/datanucleus-core.jar:lib/datanucleus-rdbms.jar:
         lib/datanucleus-jpa.jar:lib/mysql-connector-java.jar:target/classes/:.
     org.datanucleus.samples.jpa.tutorial.Main


Manually on Windows :
java -cp
lib\persistence-api.jar;lib\jdo2-api.jar;lib\datanucleus-core.jar;lib\datanucleus-rdbms.jar;
         lib\datanucleus-jpa.jar;lib\mysql-connector-java.jar;target\classes\;.
     org.datanucleus.samples.jpa.tutorial.Main


Output :

DataNucleus Tutorial with JPA
======================
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
>  Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
```

Any questions?

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

Again, you can download the sample classes from this tutorial from SourceForge.

The DataNucleus Team

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

21.2 JPA Tutorial with DB4O

DataNucleus - Tutorial for JPA using DB4O

Background

We saw in the JPA Tutorial the overall process for adding persistence to a simple application. In that tutorial we persisted the objects to an RDBMS. Here we show the differences when persisting instead to the db4o object datastore.

1. Step 1 : Design your domain/model classes as you would do normally
2. Step 2 : Define their persistence definition using Meta-Data.
3. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
4. Step 4 : Generate the database tables where your classes are to be persisted.
5. Step 5 : Write your code to persist your objects within the DAO layer.
6. Step 6 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-tutorial-*").

Step 1 : Create your domain/model classes

The model classes are exactly the same here, so please refer to the JPA Tutorial for details.

Step 2 : Define the Persistence for your classes

When defining the persistence of the classes, the only difference would be that we could omit any ORM information such as column names, lengths etc since db4o doesn't make use of that. It would, however, be perfectly safe to leave them in and DataNucleus will ignore them.

Step 3 : Instrument/Enhance your classes

In the JPA Tutorial we saw that we need to bytecode enhance our classes to be persisted. This step is identical when using db4o so please refer to the original tutorial for details.

Step 4 : Generate any schema required for your domain classes

With db4o we have no such concept as a "schema" and so this step is omitted.

Step 5 : Write the code to persist objects of your classes

We saw in the JPA Tutorial how to persist our objects using JPA API calls. This is the same when using db4o so please refer to the original tutorial for details.

Please note that DataNucleus doesn't currently support use of JPQL/SQL with *db4o* so any

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

operations for querying will throw an Exception informing you of this.

Step 6 : Run your application

We saw in the JPA Tutorial how to run our JPA-enabled application. This is very similar when persisting to db4o. The only differences are

- Put your db4o.jar in the CLASSPATH instead of the JDBC driver
- Put the **DataNucleus DB4O** JAR in the CLASSPATH instead of **DataNucleus RDBMS** JAR
- Edit *persistence.xml* to include the details for your DB4O datastore

The *persistence.xml* file could be changed to be like this

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <mapping-file>com/datanucleus/samples/jpa/tutorial/orm.xml</mapping-file>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <properties>
            <property name="datanucleus.ConnectionURL" value="db4o:file:db4o.db"/>
        </properties>
    </persistence-unit>

</persistence>
```

Any questions?

As you can see changing between an RDBMS and db4o is trivial. You don't need to change your actual classes, or persistence code at all. It's simply a change to your runtime details.

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

21.3 DataNucleus and Eclipse Dali

DataNucleus, Eclipse Dali, JPA

The Eclipse Dali project provides a powerful development environment for Java Persistence. DataNucleus does not stay behind, and permits the powerful DataNucleus persistence engine to be combined with Eclipse Dali for development.

In this (5 mins) tutorial, we use Eclipse Dali to reverse engineer a database table (ACCOUNT) and generate a persistent class (Account). The DataNucleus Eclipse plug-in is used to enhance the persistent class before running the application.

Requirements

For using the IDE, you must install Eclipse 3.2, Eclipse Dali and the DataNucleus Eclipse plug-in. For using the DataNucleus runtime, see JPA annotations.

Demo

Source Code

The source code for *org.jpox.demo.Account* class.

```
package org.jpox.demo;

import java.io.Serializable;
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Account implements Serializable {
    @Id
    @Column(name="ACCOUNT_ID")
    private BigDecimal accountId;

    private String username;

    private BigDecimal enabled;

    private static final long serialVersionUID = 1L;

    public Account() {
        super();
    }

    public BigDecimal getAccountId() {
        return this.accountId;
    }

    public void setAccountId(BigDecimal accountId) {
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        this.accountId = accountId;
    }

    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public BigDecimal getEnabled() {
        return this.enabled;
    }

    public void setEnabled(BigDecimal enabled) {
        this.enabled = enabled;
    }

}
```

The source code for *org.jpox.demo.Main* class.

```
package org.jpox.demo;

import java.math.BigDecimal;
import java.util.Random;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

public class Main
{
    public static void main(String[] args)
    {
        java.io.InputStream is =
Main.class.getClassLoader().getResourceAsStream("PMFProperties.properties");
        PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(is);
        PersistenceManager pm = pmf.getPersistenceManager();
        try
        {
            pm.currentTransaction().begin();
            Account acc = new Account();
            BigDecimal dec = new BigDecimal(new Random().nextInt());
            acc.setAccountId(dec);
            acc.setEnabled(BigDecimal.ONE);
            pm.makePersistent(acc);
            pm.currentTransaction().commit();
            System.out.println("Account "+dec+" was persisted.");
        }
        finally
        {
            if( pm.currentTransaction().isActive() )
            {
                pm.currentTransaction().rollback();
            }
            pm.close();
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

```
        }
    }
}
```

The source code for *PMFProperties.properties* file.

```
javax.jdo.PersistenceManagerFactoryClass=org.jpox.PersistenceManagerFactoryImpl
javax.jdo.option.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionURL=jdbc:oracle:thin:@127.0.0.1:1521:XE
javax.jdo.option.ConnectionUserName=test
javax.jdo.option.ConnectionPassword=password

org.jpox.autoCreateSchema=true
org.jpox.metadata.validate=false
org.jpox.autoStartMechanism=XML
org.jpox.autoCreateTables=true
org.jpox.validateTables=false
org.jpox.autoCreateColumns=true
org.jpox.autoCreateConstraints=true
org.jpox.validateConstraints=false
org.jpox.autoCreateSchema=true
org.jpox.rdbms.stringDefaultLength=255
```

The database schema model.

```
CREATE TABLE Account (
        ACCOUNT_ID NUMBER NOT NULL,
        username VARCHAR2(255),
        enabled NUMBER(1 , 0) NOT NULL
    );

ALTER TABLE Account ADD CONSTRAINT Account_PK PRIMARY KEY (ACCOUNT_ID);
```

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

Table of Contents

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

1 **Access Platform**
1.1 Welcome. 1
1.2 Getting Started. 2
1.3 Development Process. 3
1.4 What's New ?. 4
1.5 Upgrade Migration. 5
1.6 Dependencies. 7
1.7 Architecture. 9
1.8 Compatibility. 10
1.9 JDO-JPA FAQ. 11

2 **Persistence**
2.1 Bytecode Enhancement. 13
2.2 Enhancer. 16
2.3 Persistence Properties. 24
2.4 Object Lifecycle. 41
2.5 Performance Tuning. 42
2.6 Failover. 48
2.7 Security. 51
2.8 Troubleshooting. 53
2.9 Management (JMX). 58
2.10 Logging. 60
2.11 ORM Relationships. 64

3 **JDO Class Mapping**
3.1 JDO Class Mapping. 67
3.2 Java Types. 71
3.3 Datastore Identity. 77
3.4 Application Identity. 80
3.5 Nondurable Identity. 82
3.6 Primary Keys. 83
3.7 Fields/Properties. 87
3.8 Value Generation. 89
3.9 Sequences. 100

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

3.10 JDO MetaData. 104
3.10.1 JDO XML. 106
3.10.2 JDO Annotations. 136
3.10.3 JDO MetaData API. 165
3.11 Persistence Unit. 167

4 JDO O/R Mapping

4.1 ORM with JDO. 169
4.2 ORM Meta-Data. 170
4.3 Schema Mapping. 172
4.4 Datastore Identifiers. 179
4.5 Secondary Tables. 182
4.6 Embedded Objects. 185
4.7 Serialised Objects. 196
4.8 Constraints. 203
4.9 Inheritance. 208
4.10 Interfaces. 217
4.11 Objects. 221
4.12 Arrays. 225
4.13 Versioning. 230

5 JDO Relationships

5.1 Managing Relationships. 232
5.2 Cascading. 234
5.3 1-to-1. 239
5.4 1-to-N. 242
5.4.1 1-to-N (Collection). 243
5.4.2 1-to-N (Set). 254
5.4.3 1-to-N (List). 261
5.4.4 1-to-N (Map). 269

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

5.5 N-to-1. 277
5.6 M-to-N. 279
5.7 Compound Identity Relation. 282

6 JPA Class Mapping

6.1 JPA Class Mapping. 291
6.2 Java Types. 294
6.3 Application Identity. 300
6.4 Primary Keys. 302
6.5 Fields/Properties. 306
6.6 Value Generation. 308
6.7 JPA MetaData. 313
6.7.1 JPA XML MetaData. 314
6.7.2 JPA Annotations. 333
6.8 Persistence Unit. 363

7 JPA O/R Mapping

7.1 ORM with JPA. 365
7.2 Schema Mapping. 366
7.3 Datastore Identifiers. 369
7.4 Secondary Tables. 371
7.5 Serialised Objects. 373
7.6 Constraints. 375
7.7 Inheritance. 378
7.8 Arrays. 386
7.9 Versioning. 388

8 JPA Relationships

8.1 Managing Relationships. 389
8.2 Cascading. 391
8.3 1-to-1. 394
8.4 1-to-N. 397
8.4.1 1-to-N (Collection). 398
8.4.2 1-to-N (Set). 405
8.4.3 1-to-N (List). 412

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.4.4 1-to-N (Map) 419

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

8.5 N-to-1. 422
8.6 M-to-N. 424
8.7 Compound Identity Relation. 427

9 **JDO API**

9.1 JDO API. 436
9.2 Persistence Manager Factory. 437
9.3 Persistence Manager. 441
9.4 Persistence Manager Proxy. 444
9.5 Auto-Start. 445
9.6 Datastore Control. 447
9.7 Datastore Replication. 448
9.8 Transaction Types. 449
9.9 Transactions. 453
9.10 Cache. 457
9.11 Datastore Connection. 463
9.12 Attach/Detach. 465
9.13 Fetch Groups. 471
9.14 Instance Callbacks. 476
9.15 Lifecycle Listeners. 478
9.16 J2EE and JDO. 483

10 **JDO Queries**

10.1 JDO Query API. 491
10.2 Named Queries. 500
10.3 JDOQL. 502
10.4 JDOQL : Result. 507
10.5 JDOQL : Methods. 510
10.6 JDOQL : Subqueries. 514
10.7 JDOQL : In-Memory. 516
10.8 SQL. 517
10.9 JPQL. 522

11 **JPA API**

11.1 JPA API. 524

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

11.2 Entity Manager Factory. 525
11.3 Entity Manager. 527
11.4 Datastore Control. 529
11.5 Datastore Replication. 530
11.6 Transaction Types. 531
11.7 Transactions. 535
11.8 Cache. 537
11.9 Lifecycle Callbacks. 542

12 **REST API**
12.1 REST API. 544
12.2 Configuration. 545
12.3 HTTP Methods. 546

13 **JPA Queries**
13.1 JPA Query API. 549
13.2 Named Queries. 552
13.3 JPQL. 554
13.4 JPQL : Subqueries. 557
13.5 JPQL : In-Memory. 559
13.6 SQL. 560

14 **Datastores**
14.1 Datastores. 562
14.2 RDBMS. 563
14.2.1 Java Types (Spatial). 573
14.2.2 Persistence Properties. 578
14.2.3 Auto-Start. 584
14.2.4 Datastore Types. 586
14.2.5 Datasource. 593
14.2.6 Connection Pooling. 597
14.2.6.1 C3P0. 599
14.2.6.2 DBCP. 600
14.2.6.3 Proxool. 601
14.2.7 Queries. 602

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.2.7.1 JDOQL. . . . 606
14.2.7.2 JPQL. . . . 609
14.2.7.3 JDOQL : Methods. . . . 611
14.2.7.4 JDOQL2 : Methods. . . . 617
14.2.7.5 JDOQL Spatial Methods. . . . 622
14.2.8 Schema. . . . 632
14.2.9 SchemaTool. . . . 634
14.2.10 Statement Batching. . . . 642
14.2.11 Views. . . . 643
14.2.12 Datastore API. . . . 646
14.3 DB4O. . . . 650
14.3.1 Persistence Properties. . . . 652
14.3.2 Queries. . . . 654
14.3.3 Native Queries. . . . 655
14.3.4 sql4o. . . . 656
14.4 LDAP. . . . 659
14.4.1 Mapping. . . . 661
14.4.2 Relations by DN. . . . 664
14.4.3 Relations by Attribute. . . . 668
14.4.4 Relations by Hierarchy. . . . 672
14.4.5 Embedded Objects. . . . 677
14.4.6 Queries. . . . 679
14.5 Excel. . . . 680
14.5.1 Mapping. . . . 681
14.5.2 Queries. . . . 683
14.6 XML. . . . 684
14.6.1 Mapping. . . . 685
14.6.2 Queries. . . . 687
14.7 NeoDatis. . . . 688
14.7.1 Queries. . . . 690
14.7.2 Native Queries. . . . 691
14.7.3 Criteria Queries. . . . 692

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

14.8 JSON. . . . 693
14.8.1 Mapping. . . . 694
14.8.2 Queries. . . . 695
14.9 ODF. . . . 696
14.9.1 Mapping. . . . 697
14.9.2 Queries. . . . 700
14.10 AppEngine. . . . 701

15 **Available Plugins**
15.1 Core. . . . 702
15.2 Enhancer. . . . 703
15.3 JPA. . . . 704
15.4 Cache. . . . 705
15.5 REST. . . . 706
15.6 Management. . . . 707
15.7 JDO > Connector. . . . 708
15.8 Store > RDBMS. . . . 709
15.8.1 ConnectionPool. . . . 710
15.8.2 Spatial. . . . 711
15.8.3 XMLTypeOracle. . . . 714
15.9 Store > DB4O. . . . 715
15.9.1 SQL4O. . . . 716

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

15.10 Store > LDAP. 717
15.11 Store > Excel. 718
15.12 Store > XML. 719
15.13 Store > Neodatis. 720
15.14 Store > JSON. 721
15.15 Store > ODF. 722
15.16 Maven1 Plugin. 723
15.17 Maven2 Plugin. 724

16 **Plugin Mechanism**
16.1 Plugins. 725

17 **Core Extension-Points**
17.1 Java Types. 729
17.2 Store Manager. 734
17.3 AutoStart Mechanisms. 736
17.4 ClassLoader Resolvers. 739
17.5 Datastore Identity. 744
17.6 Identity Translator. 749
17.7 Annotations. 751
17.8 MetaData Handler. 753
17.9 MetaData Entity Resolver. 754
17.10 Value Generators. 755
17.11 Level 1 Cache. 759
17.12 Level 2 Cache. 761
17.13 Query Language. 764
17.14 Query Methods. 765
17.15 Class Enhancers. 769
17.16 Implementation Creator. 771
17.17 Management Server. 772
17.18 JTA Locator. 773

18 **RDBMS Extension-Points**
18.1 JDOQL - User Methods. 775
18.2 Java Type Mapping. 778

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

18.3 Datastore Mapping. . . . 784
18.4 Datastore Adapter. . . . 789
18.5 ConnectionPool. . . . 793
18.6 ConnectionProvider. . . . 796
18.7 Identifier Factories. . . . 798
18.8 SQL Methods. . . . 804
18.9 SQL Operations. . . . 806
18.10 SQL Table Namer. . . . 808
18.11 RDBMS Requests. . . . 810

19 **Access Platform Guides**

19.1 Guides. . . . 812
19.2 DataNucleus and Eclipse. . . . 813
19.3 DataNucleus and NetBeans. . . . 821
19.4 DataNucleus and Maven1. . . . 830
19.5 DataNucleus and Maven2. . . . 836
19.6 DataNucleus and Ivy. . . . 838
19.7 Enhancing with Ant. . . . 841
19.8 User Types Extension. . . . 847

20 **JDO Guides**

20.1 The JDO Tutorial. . . . 861
20.2 JDO Tutorial with DB4O. . . . 870
20.3 1-N Bidir FK Relation. . . . 872
20.4 1-N Bidir JoinTable Relation. . . . 878
20.5 M-N Relation. . . . 886
20.6 M-N Attributed Relation. . . . 895
20.7 Datastore Replication. . . . 899
20.8 Spatial Types Tutorial. . . . 902
20.9 JFire. . . . 907
20.10 Springframework. . . . 910
20.11 DAO Layer Design. . . . 919
20.12 Tapestry : VLib. . . . 925
20.13 Tapestry : Petstore. . . . 933
20.14 Web sample (LongPlay). . . . 940

©2008-2009, DataNucleus • ALL RIGHTS RESERVED

21 JPA Guides

21.1 The JPA Tutorial. 948
21.2 JPA Tutorial with DB4O. 958
21.3 DataNucleus and Eclipse Dali. 960

©2008-2009, DataNucleus • ALL RIGHTS RESERVED