

SolarMetric Kodo JDO Developers Guide

SolarMetric Kodo™ JDO Developers Guide

Copyright © 2002 SolarMetric Inc.

Table of Contents

Part I. Introduction

Chapter 1. Introduction to SolarMetric Kodo JDO

Part II. Java Data Objects

Chapter 1. Introduction

- 1.1. Intended Audience 4
- 1.2. Transparent Persistence 4

Chapter 2. Why JDO?

Chapter 3. JDO Architecture

- 3.1. JDO Exceptions 8

Chapter 4. PersistenceCapable

- 4.1. Enhancers 9
- 4.2. Persistence-Capable vs. Persistence-Aware 10
- 4.3. Restrictions on Persistent Classes 10
 - 4.3.1. Inheritance 11
 - 4.3.2. Persistent Fields 11
 - 4.3.3. Conclusions 13
- 4.4. InstanceCallbacks 13
- 4.5. JDO Identity 14
 - 4.5.1. Datastore Identity 15
 - 4.5.2. Application Identity 15
- 4.6. Conclusions 16

Chapter 5. Metadata

- 5.1. Metadata DTD 17
- 5.2. Metadata Placement 21

Chapter 6. JDOHelper

- 6.1. Persistence-Capable Operations 23
- 6.2. Lifecycle Operations 23
- 6.3. PersistenceManagerFactory Construction 26

Chapter 7. PersistenceManagerFactory

- 7.1. Obtaining a PersistenceManagerFactory 28
- 7.2. PersistenceManagerFactory Properties 29
 - 7.2.1. Connection Configuration 29
 - 7.2.2. PersistenceManagerFactory and Transaction Defaults 30
 - 7.2.3. PersistenceManager Pooling 30
- 7.3. Obtaining PersistenceManagers 31
- 7.4. Properties and Supported Options 31

Chapter 8. PersistenceManager

- 8.1. User Object Association 34
- 8.2. Configuration Properties 34
- 8.3. Transaction Association 34
- 8.4. Persistence-capable Lifecycle Management 34
- 8.5. JDO Identity Management 35
- 8.6. Extent Factory 36
- 8.7. Query Factory 36
- 8.8. Closing 36

Chapter 9. Transaction

- 9.1. Transaction Types 37
- 9.2. The JDO Transaction Interface 38

Chapter 10. Extent

Chapter 11. Query

- 11.1. Required Query Elements 41
- 11.2. Optional Query Elements 41
- 11.3. JDOQL 42
- 11.4. Executing Queries 45

| | |
|--|----|
| 11.5. Query Compilation | 45 |
| Part III. Kodo JDO Reference Guide | |
| Chapter 1. Introduction | |
| 1.1. Intended Audience | 47 |
| Chapter 2. Configuration Framework | |
| 2.1. JDO Standard Properties | 48 |
| 2.1.1. javax.jdo.PersistenceManagerFactoryClass | 49 |
| 2.1.2. javax.jdo.option.Optimistic | 49 |
| 2.1.3. javax.jdo.option.Multithreaded | 49 |
| 2.1.4. javax.jdo.option.IgnoreCache | 49 |
| 2.1.5. javax.jdo.option.RetainValues | 50 |
| 2.1.6. javax.jdo.option.RestoreValues | 50 |
| 2.1.7. javax.jdo.option.NontransactionalRead | 50 |
| 2.1.8. javax.jdo.option.NontransactionalWrite | 50 |
| 2.1.9. javax.jdo.option.ConnectionURL | 51 |
| 2.1.10. javax.jdo.option.ConnectionUserName | 51 |
| 2.1.11. javax.jdo.option.ConnectionPassword | 51 |
| 2.1.12. javax.jdo.option.ConnectionDriverName | 51 |
| 2.1.13. javax.jdo.option.ConnectionFactoryName | 51 |
| 2.1.14. javax.jdo.option.ConnectionFactory2Name | 52 |
| 2.1.15. javax.jdo.option.MinPool | 52 |
| 2.1.16. javax.jdo.option.MaxPool | 52 |
| 2.1.17. javax.jdo.option.MsWait | 52 |
| 2.2. Kodo JDO Properties | 53 |
| 2.2.1. com.solarmetric.kodo.CacheReferenceSize | 53 |
| 2.2.2. com.solarmetric.kodo.DataCacheClass | 53 |
| 2.2.3. com.solarmetric.kodo.ConnectionProperties | 53 |
| 2.2.4. com.solarmetric.kodo.DataCacheProperties | 54 |
| 2.2.5. com.solarmetric.kodo.DefaultFetchThreshold | 54 |
| 2.2.6. com.solarmetric.kodo.DefaultFetchBatchSize | 54 |
| 2.2.7. com.solarmetric.kodo.EnableQueryExtensions | 55 |
| 2.2.8. com.solarmetric.kodo.LicenseKey | 55 |
| 2.2.9. com.solarmetric.kodo.PersistenceManagerClass | 55 |
| 2.2.10. com.solarmetric.kodo.PersistenceManagerProperties | 55 |
| 2.2.11. com.solarmetric.kodo.ProxyManagerClass | 56 |
| 2.2.12. com.solarmetric.kodo.ProxyManagerProperties | 56 |
| 2.2.13. com.solarmetric.kodo.QueryFilterListeners | 56 |
| 2.2.14. com.solarmetric.kodo.UseSoftTransactionCache | 56 |
| 2.2.15. com.solarmetric.kodo.PersistentTypes | 57 |
| 2.2.16. com.solarmetric.kodo.impl.jdbc.ConnectionTestTimeout | 57 |
| 2.2.17. com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass | 57 |
| 2.2.18. com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass | 57 |
| 2.2.19. com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderProperties | 58 |
| 2.2.20. com.solarmetric.kodo.impl.jdbc.DictionaryClass | 58 |
| 2.2.21. com.solarmetric.kodo.impl.jdbc.DictionaryProperties | 59 |
| 2.2.22. com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping | 61 |
| 2.2.23. com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass | 61 |
| 2.2.24. com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties | 62 |
| 2.2.25. com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass | 62 |
| 2.2.26. com.solarmetric.kodo.impl.jdbc.StatementCacheMaxSize | 62 |
| 2.2.27. com.solarmetric.kodo.impl.jdbc.StatementExecutionTimeout | 63 |
| 2.2.28. com.solarmetric.kodo.impl.jdbc.SynchronizeSchema | 63 |
| 2.2.29. com.solarmetric.kodo.impl.jdbc.TransactionIsolation | 63 |
| 2.2.30. com.solarmetric.kodo.impl.jdbc.UsePreparedStatements | 64 |
| 2.2.31. com.solarmetric.kodo.impl.jdbc.UseBatchedStatements | 64 |
| 2.2.32. com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure | 64 |
| 2.2.33. com.solarmetric.kodo.ee.ManagedRuntimeClass | 64 |
| 2.2.34. com.solarmetric.kodo.ee.ManagedRuntimeProperties | 65 |

| | |
|---|-----|
| 2.3. Logging Framework | 65 |
| Chapter 3. Creating Persistent Classes | |
| 3.1. Application Identity Class Generation | 68 |
| 3.2. Enhancement | 68 |
| 3.3. Auto-Generating Classes from a Schema | 69 |
| 3.3.1. Schema File DTD | 71 |
| 3.4. Smart Proxies | 72 |
| Chapter 4. Metadata | |
| 4.1. Dependency Extensions | 73 |
| 4.2. Class-level Object-Relational Mapping Extensions | 73 |
| 4.3. Field-level Object-Relational Mapping Extensions | 74 |
| 4.4. Extensions Under Application Identity | 76 |
| 4.5. Examples | 76 |
| 4.6. Multi-table Inheritance Mapping | 80 |
| 4.7. Generating Default JDO Metadata | 80 |
| Chapter 5. JDBC Configuration | |
| 5.1. Supported Databases | 82 |
| 5.2. Accessing Multiple Databases | 82 |
| 5.3. Connection Management | 82 |
| 5.4. Large Result Sets | 83 |
| 5.5. Schema Manipulation | 84 |
| Chapter 6. Standard Features | |
| 6.1. Manipulating Datastore Identity Objects | 86 |
| 6.2. ExtentImpl | 86 |
| 6.3. PersistenceManager Extension | 86 |
| 6.4. Custom Proxies | 86 |
| 6.5. Access to SQL Connections | 87 |
| 6.6. PersistenceManagerImpl.evictAll() API extensions | 87 |
| 6.7. Custom Class Indicators | 87 |
| Chapter 7. Enterprise Features | |
| 7.1. EEPersistenceManager object listeners | 89 |
| 7.2. EEPersistenceManager transaction listeners | 89 |
| 7.3. Datastore Cache | 89 |
| 7.3.1. Overview of Kodo JDO Datastore Caching | 89 |
| 7.3.2. Kodo JDO Cache Usage | 90 |
| 7.3.3. Cache Extension | 93 |
| 7.3.4. Important notes about the DataCache | 93 |
| 7.3.5. Known issues and limitations | 94 |
| 7.4. Query Extensions | 94 |
| 7.4.1. Using extensions in queries | 94 |
| 7.4.2. Included Kodo Extensions | 95 |
| 7.4.3. Writing custom extensions | 95 |
| 7.4.4. Configuring Extensions | 96 |
| 7.5. Fetch Groups | 97 |
| 7.5.1. Normal Default Fetch Group Behavior | 97 |
| Chapter 8. Additional Features | |
| 8.1. Custom data processing | 99 |
| 8.2. Custom data requests | 99 |
| Chapter 9. Third Party Integration Features | |
| 9.1. Overview of Third Party Integration features in Kodo | 101 |
| 9.2. Apache Ant | 101 |
| 9.2.1. Common Ant Configuration Options | 101 |
| 9.2.2. JDOEnhancer Ant Task | 102 |
| 9.2.3. SchemaTool Ant Task | 102 |
| 9.3. XDoclet | 103 |
| 9.4. Borland JBuilder | 106 |
| 9.4.1. Installing Kodo into JBuilder | 106 |
| 9.4.2. Kodo Configuration from JBuilder | 107 |

| | |
|---|-----|
| 9.4.3. Creating and building JDO projects in JBuilder | 107 |
| 9.4.4. Editing JDO Metadata from JBuilder | 107 |
| 9.4.5. Running the SchemaTool from JBuilder | 107 |
| 9.4.6. JBuilder Project Sample | 108 |
| 9.5. Sun ONE Studio / NetBeans IDE | 108 |
| 9.5.1. Before Installing Kodo into the IDE | 108 |
| 9.5.2. Installing Kodo into the IDE | 108 |
| 9.5.3. Configuring the Kodo Module | 109 |
| 9.5.4. Kodo Template Wizards | 109 |
| 9.5.5. JDO DataObject | 110 |
| 9.5.6. Kodo Integration into the Build Process | 110 |
| 9.5.7. SunONE / NetBeans Sample | 110 |
| 9.6. Eclipse / WebSphere Studio Integration | 110 |
| 9.6.1. Installing the Kodo Eclipse Plugin | 111 |
| 9.6.2. Configuring the Plugin | 111 |
| 9.6.3. Using Kodo in Eclipse IDEs | 111 |
| 9.6.4. Eclipse Sample | 112 |
| Chapter 10. Enterprise Integration | |
| 10.1. Overview of Enterprise Integration features in Kodo | 113 |
| 10.2. Using Kodo JDO via the Java Connector Architecture | 114 |
| 10.2.1. Overview of the JCA | 114 |
| 10.2.2. Deploying on JBoss 3.0 | 114 |
| Chapter 11. Kodo JDO Implementation Notes | |
| 11.1. PersistenceCapable.jdoFlags field and fields in the Default Fetch Group | 116 |
| 11.2. Optimistic locking mechanism | 116 |
| Chapter 12. Optimization Techniques | |
| Part IV. Kodo JDO Tutorial | |
| Introduction to the Kodo JDO Tutorial | |
| 1.1. Tutorial Requirements | 121 |
| Chapter 1. The Pet Shop | |
| 1.1. Included Files | 122 |
| 1.2. Important Utilities | 123 |
| Chapter 2. Getting Started | |
| Chapter 3. Inventory Maintenance | |
| Chapter 4. Inventory Growth | |
| Chapter 5. Behavioral Analysis | |
| Chapter 6. Extra Features | |
| Part V. Reverse Mapping Tool Tutorial | |
| Introduction to the Reverse Mapping Tool Tutorial | |
| 1.1. Reverse Mapping Tool Tutorial Requirements | 138 |
| Chapter 1. Magazine Shop | |
| Chapter 2. Setup | |
| 2.1. Tutorial Files | 140 |
| 2.2. Important Utilities | 140 |
| Chapter 3. Generating Persistent Classes | |
| Chapter 4. Using the Finder | |
| Appendix A. Development and Runtime Libraries | |
| Appendix B. JDO Resources | |
| Appendix C. Optional JDO Features | |
| Appendix D. SQL Types | |
| Appendix E. Release Notes | |
| Appendix F. Known Bugs and Limitations | |
| Appendix G. Supported Databases | |
| G.1. Example properties for Pointbase JDBC driver (Pointbase) | 166 |
| G.2. Example properties for InstantDB JDBC driver (InstantDB) | 167 |
| G.3. Example properties for Cloudscape JDBC driver (Cloudscape) | 167 |
| G.4. Example properties for DB2 JDBC driver (DB2) | 168 |
| G.5. Example properties for Oracle JDBC driver (Oracle) | 168 |

| | |
|--|-----|
| G.6. Example properties for PostgreSQL Native Driver (PostgreSQL) | 168 |
| G.7. Example properties for SQLServer (Microsoft SQL Server) | 169 |
| G.8. Example properties for jConnect (Sybase Adaptive Server Enterprise) | 169 |
| G.9. Example properties for Hypersonic (Hypersonic Database Engine) | 170 |
| G.10. Example properties for MySQL Driver (MySQL) | 170 |

List of Tables

| | |
|---|-----|
| Table D.1. SQL Type Mappings | 148 |
| Table D.2. SQL Type Mappings II | 148 |
| Table D.3. SQL Type Mappings III | 148 |
| Table G.1. Supported Databases and JDBC Drivers | 166 |

List of Examples

| | |
|--|-----|
| Example 4.1. PersistenceCapable Class | 9 |
| Example 4.2. Accessing Mutable Persistent Fields | 12 |
| Example 5.1. Basic Structure of Metadata Documents | 17 |
| Example 5.2. Metadata Class Listings | 19 |
| Example 5.3. Complete Metadata Document | 20 |
| Example 6.1. Obtaining a PersistenceManagerFactory | 27 |
| Example 9.1. Grouping Operations with Transactions | 38 |
| Example 11.1. Basic Query | 43 |
| Example 11.2. Result Ordering and Method Calls | 43 |
| Example 11.3. Mathematical Operations and Relation Traversal | 44 |
| Example 11.4. Precedence and Logical Operators | 44 |
| Example 11.5. Imports and Parameters | 44 |
| Example 11.6. Collections | 44 |
| Example 11.7. Variables | 45 |
| Example 2.1. Specifying Connection Properties | 54 |
| Example 2.2. Specifying DataSource Properties | 54 |
| Example 2.3. Example log4j.properties file for moderately verbose logging | 66 |
| Example 2.4. Example log4j.properties file for disabled logging | 66 |
| Example 2.5. Example log4j.properties file for debugging logging | 67 |
| Example 3.1. Using the Application Identity Tool | 68 |
| Example 3.2. Using the Kodo JDO Enhancer | 69 |
| Example 4.1. Mapping Classes to an Existing Schema | 76 |
| Example 4.2. Extensions Under Application Identity | 78 |
| Example 5.1. Configuring custom dictionary properties | 82 |
| Example 5.2. Properties for using a custom DataSource | 83 |
| Example 5.3. Using Random Access Query Results in a Portable Fashion | 83 |
| Example 5.4. Using the Kodo JDO Schematool | 85 |
| Example 6.1. Obtaining a java.sql.Connection object from the PersistenceManager | 87 |
| Example 6.2. IntegerSubclassProvider example | 88 |
| Example 7.1. Pinning an object into the DataCache | 90 |
| Example 7.2. Unpinning an object from the DataCache | 90 |
| Example 7.3. Evicting an object from the DataCache | 90 |
| Example 7.4. Configuring a PersistenceManagerFactory to use a JMS cache update mechanism | 91 |
| Example 7.5. Configuring a PersistenceManagerFactory to use a Tangosol cache for distributed cache needs | 91 |
| Example 7.6. Configuring a PersistenceManagerFactory to use a TCP cache update mechanism | 92 |
| Example 7.7. Configuring a PersistenceManagerFactory to use a UDP cache update mechanism | 93 |
| Example 7.8. Using Kodo's StringContains extension | 95 |
| Example 7.9. Filter extensions usage example | 96 |
| Example 9.1. Using the <config> tag in an ant build.xml file | 101 |
| Example 9.2. Using the <classpath> tag in an ant build.xml file | 102 |
| Example 9.3. Invoking the JDOEnhancer from an Ant build.xml file | 102 |
| Example 9.4. Invoking the SchemaTool from an Ant build.xml file | 103 |
| Example 9.5. Invoking the JDO Metadata generator from an Ant build.xml file | 103 |
| Example 9.6. Source code comments for automatic JDO Metadata generation | 104 |
| Example 10.1. Binding a PersistenceManagerFactory into JNDI via a WebLogic startup class | 113 |
| Example 10.2. Looking up the PersistenceManagerFactory in JNDI | 114 |
| Example 12.1. Explicitly closing resources | 117 |

Part I. Introduction

Chapter 1. Introduction to SolarMetric Kodo JDO

This document provides an introduction to Sun's Java Data Objects (JDO) specification and an overview of the basic setup and use of the Kodo JDO implementation for relational databases.

- To familiarize yourself with the concepts involved in the JDO specification, refer to the JDO Overview.
- To quickly get started with Kodo JDO, take the Kodo JDO Tutorial.
- For a detailed reference on the Kodo JDO implementation, refer to the Kodo JDO Reference Guide.

Part II. Java Data Objects

Chapter 1. Introduction

Java Data Objects (JDO) is a specification from Sun Microsystems for the transparent persistence of Java objects to any transactional data store. This document provides an overview of JDO. The information presented applies to all JDO implementations, unless otherwise noted.

1.1. Intended Audience

This document is intended for developers who want to learn about JDO in order to use it in their applications. It assumes that you have a strong knowledge of Java and object-oriented concepts, and a familiarity with the eXtensible Markup Language (XML). The chapters on using JDO in a managed environment and using JDO with Enterprise Java Beans (EJBs) further assume that you have a firm grasp of Java Enterprise Edition (J2EE) architecture and of the EJB specification, respectively. This document does *not*, however, assume any experience with database programming or the manipulation of persistent data in general.

If your goal is to understand every nuance of JDO or to create your own JDO implementation, then you should skip this document and go directly to the official JDO specification, available from Sun Microsystems.

1.2. Transparent Persistence

Persistent data is information that can outlive the program that creates it. The majority of complex programs use persistent data: GUI applications need to store user preferences across program invocations, web applications track user movements and orders over long periods of time, etc.

Transparent persistence is the storage and retrieval of persistent data with little or no work from you, the developer. For example, Java serialization is a form of transparent persistence because it can be used to persist Java objects directly to a file with very little effort. Serialization's capabilities as a transparent persistence mechanism pale in comparison to those provided by JDO, however. The next chapter compares JDO to serialization and other available persistence mechanisms.

Chapter 2. Why JDO?

Java developers who need to store and retrieve persistent data already have several options available to them: serialization, JDBC, object-relational mapping tools, object databases, and entity EJBs. Why introduce yet another persistence framework? The answer to this question is that each of the aforementioned persistence solutions has severe limitations. JDO attempts to overcome these limitations.

- *Serialization* is Java's built-in mechanism for transforming an object graph into a series of bytes, which can then be sent over the network or stored in a file. Serialization is very easy to use, but it is also very limited. It must store and retrieve the entire object graph at once, making it unsuitable for dealing with large amounts of data. It cannot undo changes that are made to objects if an error occurs while updating information, making it unsuitable for applications that require strict data integrity. Multiple threads or programs cannot read and write the same serialized data concurrently without conflicting with each other. It provides no query capabilities. All these factors make serialization useless for all but the most trivial persistence needs.
- Many developers use the *Java Database Connectivity* (JDBC) APIs to manipulate persistent data in relational databases. JDBC overcomes most of the shortcomings of serialization: it can handle large amounts of data, has mechanisms to ensure data integrity, supports concurrent access to information, and has a sophisticated query language in SQL. Unfortunately, JDBC does not duplicate serialization's ease of use. The relational paradigm used by JDBC was not designed for storing objects, and therefore forces you to either abandon object-oriented programming for the portions of your code that deal with persistent data, or to find a way of mapping object-oriented concepts like inheritance to relational databases yourself.
- Several software companies created frameworks to perform the mapping between objects and relational database tables for you. These *object-relational mapping* products allow you to focus on the object model and not concern yourself with the mismatch between the object-oriented and relational paradigms. Unfortunately, each object-relational mapping product has its own set of APIs. Your code becomes tied to the proprietary interfaces of a single vendor. If the vendor raises prices or fails to fix show-stopping bugs, you cannot switch to another product without rewriting all of your persistence code. This is referred to as vendor lock-in.
- Rather than map objects to relational databases, some software companies developed a new form of database designed specifically to store objects. These *object databases* are often much easier to use than object-relational mapping software. The Object Database Management Group (ODMG) was formed to create a standard API for accessing object databases; few object database vendors, however, comply with the ODMG's recommendations. Thus, vendor lock-in plagues object databases as well. Many companies are also hesitant to switch from tried-and-true relational systems to the relatively new object database technology. Fewer data-analysis tools are available for object database systems, and there are vast quantities of data already stored in older relational databases. For all of these reasons and more, object databases have not caught on as well as their creators hoped.
- The Enterprise Edition of the Java platform introduced entity Enterprise Java Beans (EJBs). Entity EJBs are components that represent persistent information in a data store. Like object-relational mapping solutions, entity EJBs provide an object-oriented view of persistent data. Unlike object-relational software, however, entity EJBs are not limited to relational databases; the persistent information they represent may come from an Enterprise Information System (EIS) or other storage device. Also, EJBs use a strict standard, making them portable across vendors. Unfortunately, the EJB standard is somewhat limited in the object-oriented concepts it can represent. Advanced features like inheritance, polymorphism, and complex relations are absent. Additionally, EJBs are difficult to code, and they require heavyweight and often expensive application servers to run. EJBs, especially session and message-driven beans, do have other advantages, however, and so the JDO specification details how JDO can integrate with them.

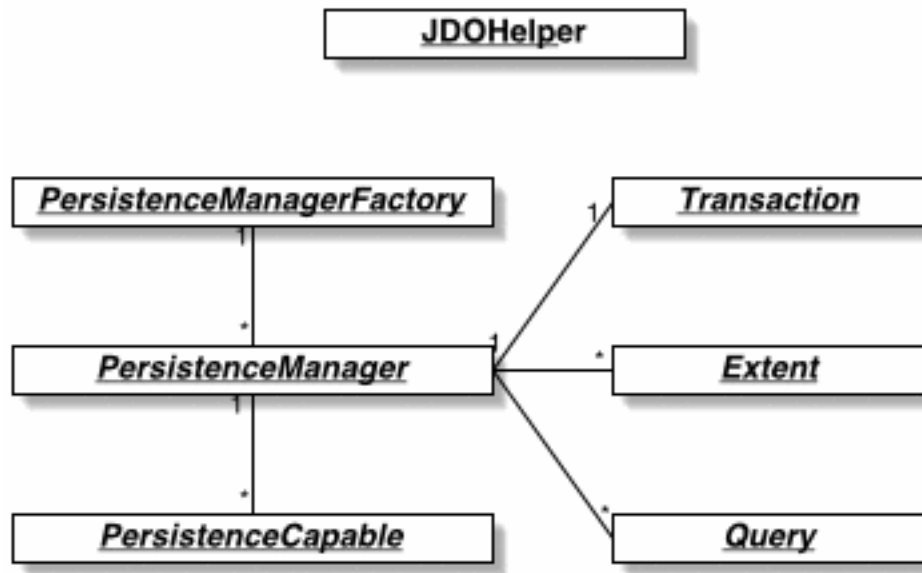
JDO combines many of the best features from each of the persistence mechanisms listed above. Creating persistent classes under JDO is as simple as creating serializable classes. JDO supports the large data sets, data consistency, concurrent use, and query capabilities of JDBC. Like object-relational software and object databases, it allows the use of advanced object-oriented concepts such as inheritance. It avoids vendor lock-in by relying on a strict specification like entity EJBs. Also like entity EJBs, JDO does not prescribe any specific back-end data store. JDO implementations might store objects in relational databases, object databases, flat files, or any other persistent storage device.

Note

Kodo JDO stores objects in relational databases using JDBC.

JDO is not ideal for every application. For many applications, though, it provides an exciting alternative to other persistence mechanisms.

Chapter 3. JDO Architecture

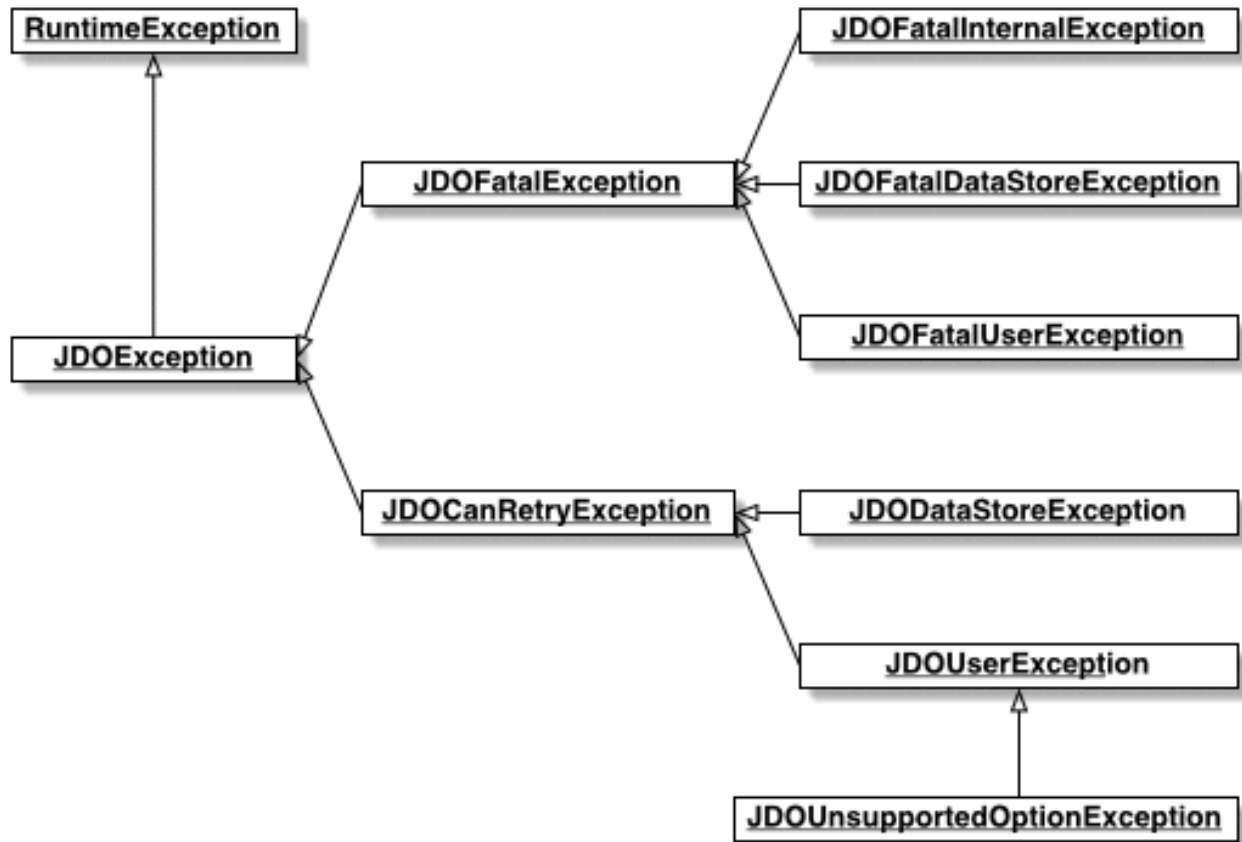


The diagram above illustrates the relationships between the primary components of the JDO architecture. These components are:

- *JDOHelper*. The `javax.jdo.JDOHelper` contains static helper methods to query the lifecycle state of persistent objects and to create concrete `PersistenceManagerFactory` instances in a vendor-neutral fashion.
- *PersistenceManagerFactory*. The `javax.jdo.PersistenceManagerFactory` is a factory for `PersistenceManagers`.
- *PersistenceManager*. The `javax.jdo.PersistenceManager` is the primary JDO interface used by applications. Each `PersistenceManager` manages a set of persistent objects and has APIs to persist new objects and delete existing persistent objects. There is a one-to-one relationship between a `PersistenceManager` and a `Transaction`. `PersistenceManagers` also act as factories for `Extent` and `Query` instances.
- *PersistenceCapable*. User-defined persistent classes must implement the `javax.jdo.spi.PersistenceCapable` interface. Most JDO implementations provide an *enhancer* that transparently adds the code to implement this interface to each persistent class. You should never use the `PersistenceCapable` interface directly.
- *Transaction*. Each `PersistenceManager` has a one-to-one relation with a single `javax.jdo.Transaction`. Transactions allow operations on persistent data to be grouped into units of work that either completely succeed or completely fail, leaving the data store in its original state. These all-or-nothing operations are important for maintaining data integrity.
- *Extent*. The `javax.jdo.Extent` is a logical view of all the objects of a particular class that exist in the data store. Extents can be configured to also include subclasses. Extents are obtained from a `PersistenceManager`.
- *Query*. The `javax.jdo.Query` interface is implemented by each JDO vendor to translate expressions in the Java Data Objects Query Language (JDOQL), which is based on Java boolean expressions, into the native query language of the data store. You obtain `Query` instances from a `PersistenceManager`.

The majority of the remainder of this document is dedicated to exploring each of these components. We present them in roughly the order that you will use them as you develop your application.

3.1. JDO Exceptions



The above diagram outlines the JDO exception architecture. Runtime exceptions such as `NullPointerException`s and `IllegalArgumentException`s aside, JDO components throw nothing but `JDOExceptions` of one type or another.

The JDO exception hierarchy should be self-explanatory. Consult the JDO Javadoc for details.

Chapter 4. PersistenceCapable

All user-defined persistent classes implement the `javax.jdo.spi.PersistenceCapable` interface. This interface contains many complex methods that enable the JDO implementation to manage the persistent fields of class instances. Fortunately, you do not have to implement this interface yourself in order to create persistent classes. In fact, writing a persistent class in JDO is usually no different than writing any other class. There are no special parent classes to extend from, field types to use, or methods to write. This is one important way in which JDO makes persistence completely transparent to you, the developer.

Example 4.1. PersistenceCapable Class

```
package org.mag;

/**
 * Example persistent class. Notice that it looks exactly like any other
 * class. JDO makes writing persistent classes completely transparent.
 */
public class Magazine
{
    private String    isbn;
    private String    title;
    private Set       articles = new HashSet ();
    private Date      copyright;
    private Company   publisher;

    public Magazine (String title, String isbn)
    {
        this.title = title;
        this.isbn = isbn;
    }

    public void publish (Company publisher, Date copyright)
    {
        if (copyright == null)
            copyright = new Date ();

        this.publisher = publisher;
        publisher.addMagazine (this);
        this.copyright = copyright;
    }

    public void addArticle (Article article)
    {
        articles.add (article);
    }

    // etc...
}
```

4.1. Enhancers

In order to shield you from the intricacies of the `PersistenceCapable` interface, most JDO implementations provide an *enhancer*. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. Enhancers generally come in two forms: source enhancers and bytecode enhancers.

Source enhancers change the source code in the `.java` files defining your classes to implement the `PersistenceCapable` interface. This approach has the advantage of letting you see the changes that are made to your classes; however, it has several disadvantages as well:

- You must have access to the source code of all of the classes you wish to enhance.
- If you use a source version control system such as CVS, you must be careful to only commit the unenhanced versions of your source files.
- Parsing and enhancing source files is relatively slow.
- During debugging, the line numbers reported in your stack traces will not correspond to the correct lines in your original source files.

Bytecode enhancers, on the other hand, operate on your `.class` files. They post-process the bytecode generated by your Java compiler, adding the necessary fields and methods to implement the `PersistenceCapable` interface. Bytecode enhancers overcome the difficulties associated with source enhancers:

- You do not need access to the source files of the classes you wish to make persistent.
- There is no problem with source version control systems, because your source files are not altered.
- Parsing and enhancing bytecode is fast.
- During debugging, the line numbers reported in your stack traces will correspond exactly to the correct lines in your original source files.

Even with all of these advantages, many developers feel uncomfortable with bytecode enhancement. Thus, the JDO specification does not mandate a particular form of enhancement; each JDO implementation is free to choose the form that it feels best suits its users.

Note

Kodo JDO uses bytecode enhancement.

Regardless of type, all JDO enhancers are required to be binary-compatible with each other. This means that the final enhanced class can be used not only by the JDO implementation whose enhancer created it, but by any other JDO implementation as well. The binary compatibility requirement ensures that you can package and ship persistent classes to other developers without worrying about what JDO vendor they use. It also means that you can switch JDO vendors without even recompiling your persistent classes.

4.2. Persistence-Capable vs. Persistence-Aware

Classes that have been enhanced to implement the `PersistenceCapable` interface are referred to as *persistence-capable* classes. Classes that directly access public or protected persistent fields of persistence-capable classes are called *persistence-aware*. Persistence-aware classes must also be enhanced -- each time a persistence-aware class directly accesses a persistent field of a persistence-capable class, the enhancer adds code to notify the JDO implementation that the field in question is about to be read or written. This enables the JDO implementation to synchronize the field's value with the data store as needed. Unless the persistence-aware class is also persistence-capable, the enhancer does not add code to make it implement the `PersistenceCapable` interface.

Generally, it is best to keep all of your persistent fields private, or protected but only accessed by subclasses. In addition to the standard arguments in favor of state encapsulation, this approach avoids the hassle of tracking which non-persistent classes must be enhanced as persistence-aware because they happen to access a public or protected field of some persistent class.

4.3. Restrictions on Persistent Classes

There are very few restrictions placed on persistent classes. Still, it never hurts to familiarize yourself with exactly what JDO does and does not support.

4.3.1. Inheritance

JDO fully supports inheritance in persistent classes. It allows persistent classes to inherit from non-persistent classes, persistent classes to inherit from other persistent classes, and non-persistent classes to inherit from persistent classes. It is even possible to form inheritance hierarchies in which persistence "skips" generations. There are, however, a few important limitations:

- Persistent classes cannot inherit from certain natively-implemented system classes such as `java.net.Socket` and `java.lang.Thread`.
- If a persistent class inherits from a non-persistent class, the fields of the non-persistent superclass cannot be persisted.
- All classes in an inheritance tree must use the same JDO identity type. If they use application identity, they must use the same identity class. We will cover JDO identity shortly.

4.3.2. Persistent Fields

JDO manages the state of all persistent fields. Before you access a field, JDO makes sure that it has been loaded from the data store. When you set a field, JDO records that it has changed so that the new value will be persisted. This allows you to treat the field in exactly the same way you treat any other field -- another aspect of JDO's transparent persistence.

JDO includes built-in support for most common field types. These types can be roughly divided into three categories: immutable types, mutable types, and relations.

Immutable types, once created, cannot be changed. The only way to alter a persistent field of an immutable type is to assign a new value to the field. JDO supports the following immutable types for persistent fields:

- All primitives (`int`, `float`, `byte`, etc)
- All primitive wrappers (`java.lang.Integer`, `java.lang.Float`, `java.lang.Byte`, etc)
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.util.Locale`

Persistent fields of *mutable* types can be altered without assigning the field a new value. Mutable types can be modified directly through their own methods. The JDO specification requires that implementations support the following mutable field type:

- `java.util.Date`
- `java.util.HashSet`

Note

Most JDO implementations support many other mutable field types as well. Kodo JDO supports the following:

- `java.util.List`
- `java.util.ArrayList`
- `java.util.LinkedList`
- `java.util.Vector`
- `java.util.Set`
- `java.util.SortedSet`
- `java.util.TreeSet`
- `java.util.Map`
- `java.util.HashMap`
- `java.util.SortedMap`
- `java.util.TreeMap`
- `java.util.Hashtable`
- `java.util.Properties`

Most implementations do not allow you to persist nested mutable types, such as Maps of `LinkedList`s.

JDO implementations support mutable fields by transparently replacing the field value with an instance of a special subclass of the field's declared type. For example, if your persistent object has a field containing a `java.util.Date`, the JDO implementation will transparently replace the value of that field at runtime with some vendor-specific `Date` subclass -- call it `JDODate`. The job of this subclass is to track modifications to the field. Thus the `JDODate` class will override all mutator methods of `Date` to notify the JDO implementation that the field's value has been changed. The JDO implementation then knows to write the field's new value to the data store at the next opportunity.

Of course, when you develop and use persistent classes, this is all transparent. You continue to use the standard methods of mutable fields as you normally would. It is important to know how support for mutable fields is implemented, however, in order to understand why JDO has such trouble with arrays. JDO allows you to use persistent array fields, and it automatically detects when these fields are assigned a new array value or set to `null`. Because arrays cannot be subclassed, however, JDO cannot detect when new values are written to array indexes. If you set an index of a persistent array, you must explicitly tell the JDO implementation you have changed the array field; this is referred to as "dirtying" the field. Dirtying is accomplished through the `JDOHelper`'s `makeDirty` method.

Example 4.2. Accessing Mutable Persistent Fields

```
/**
 * Example demonstrating the use of mutable persistent fields in JDO.
 * Assume Person is a persistent class.
 */
public void addChild (Person parent, Person child)
{
    // can modify most mutable types directly; JDO tracks
```

```

// the modifications for you
Date lastUp = parent.getLastUpdated ();
lastUp.setTime (System.currentTimeMillis ());
Collection children = parent.getChildren ();
children.add (child);
child.setParent (parent);

// arrays need explicit dirtying if they are modified,
// but not if the field is reset
parent.setObjectArray (new Object[0]);
child.getObjectArray ()[0] = parent;
JDOHelper.makeDirty (child, "objectArray");
}

```

As the example above illustrates, JDO supports relations between persistent objects in addition to the standard Java types covered so far. All JDO implementations should allow user-defined persistent classes and collections of user-defined persistent classes as persistent field types. The exact collection classes you can use to hold persistent relations will depend on which mutable field types the implementation supports. Some JDO implementations may also allow map fields in which the keys, values, or both are relations to other persistent objects. Again, the exact types of maps allowed depend on the implementation's mutable field type support.

Note

Kodo JDO supports user-defined persistent classes for map values, but it only allows immutable types for map keys.

Most JDO implementations also have some support for fields whose concrete class is not known. Fields declared as type `java.lang.Object` or as a user-defined interface type fall into this category. Because the information on these fields is so limited, though, there may be limitations placed on them. For example, they may be impossible to query, and loading and/or storing them may be inefficient.

Note

Kodo JDO supports persistent fields of an unknown type by serializing the field value and storing it as a sequence of bytes.

4.3.3. Conclusions

This section detailed all of the restrictions JDO places on persistent classes. While it may seem like a lot of information was presented, you will seldom find yourself hindered by these restrictions in practice. Additionally, there are often ways of using JDO's other features to circumvent any limitations you run into. The next section presents a powerful JDO feature that is particularly useful for this purpose.

4.4. InstanceCallbacks

Your persistent classes can implement the `javax.jdo.InstanceCallbacks` interface to receive callbacks when certain JDO lifecycle events take place. This interface consists of four methods:

- The `jdoPostLoad` method is called by the JDO implementation after the default fetch group fields of your class have been loaded from the data store. Default fetch groups are explained in the section on JDO metadata; for now think of the default fetch group as all of the immutable fields of the object. No other persistent fields can be accessed in this method.

`jdoPostLoad` is often used to initialize non-persistent fields whose values depend on the values of persistent

fields. For example, suppose you need to persist a `java.net.InetAddress`, which is not directly supported by JDO. You could use a persistent `String` field to hold the hostname of the address, and then in `jdoPostLoad` you could use this string to create the actual `InetAddress` instance and cache it in a non-persistent field of your object.

- `jdoPreStore` is called just before the persistent values in your object are flushed to the data store. You can access all persistent fields in this method.

`jdoPreStore` is the complement to `jdoPostLoad`. While `jdoPostLoad` is most often used to initialize non-persistent values from persistent data, `jdoPreStore` is usually used to set persistent fields with information cached in non-persistent ones. Returning to our `InetAddress` example, `jdoPreStore` would be used to retrieve the host name from the non-persistent `InetAddress` field and store it in the persistent `String` field.

- The `jdoPreClear` method is called before the persistent fields of your object are cleared. JDO implementations clear the persistent state of objects for several reasons, most of which will be covered later in this document. `jdoPreClear` can be used to clear non-persistent cached data and null relations to other objects. You should not access the values of persistent fields in this method.
- `jdoPreDelete` is called before an object is deleted from the data store. Access to persistent fields is valid within this method. You might implement privately-owned relations by using this method to delete other related objects.

Unlike the `PersistenceCapable` interface, you must implement the `InstanceCallbacks` interface explicitly if you want to receive lifecycle callbacks.

4.5. JDO Identity

Java recognizes two forms of object identity: numeric identity and qualitative identity. If two references are *numerically* identical, then they refer to the same JVM instance in memory. You can test for this using the `==` operator. *Qualitative* identity, on the other hand, relies on some user-defined criteria to determine whether two objects are "equal". You test for qualitative identity using the `equals` method. By default, this method simply relies on numerical identity.

JDO introduces another form of object identity, called JDO identity. JDO identity tests whether two persistent objects represent the same state in the data store.

You can obtain the JDO identity object from a persistent instance through the `JDOHelper`'s `getObjectId` method. If two JDO identity objects compare equal using the `equals` method, then the two corresponding persistent objects represent the same state in the data store.

If you are dealing with a single `PersistenceManager`, then there is an even easier way to test whether two persistent object references represent that same state in the data store: use the `==` operator. JDO requires that each `PersistenceManager` maintain only one JVM object to represent each unique data store record. Thus, JDO identity is equivalent to numerical identity within a `PersistenceManager`'s cache of managed objects. This is referred to as the *uniqueness requirement*.

The uniqueness requirement is extremely important -- without it, it would be impossible to maintain data integrity. Think of what could happen if two different objects of the same `PersistenceManager` were allowed to represent the same persistent data. If you made different modifications to each of these objects, which set of changes should be written to the data store? How would your application logic handle seeing two different "versions" of the same data? Thanks to the uniqueness requirement, these questions do not have to be answered.

There are three types of JDO identity, but only two of them are important to most applications: datastore identity and application identity. The majority of JDO implementations support datastore identity at a minimum; many support application identity as well.

Note

Kodo JDO supports both datastore and application identity.

4.5.1. Datastore Identity

Datastore identity is managed by the JDO implementation. It is independent of the values of your persistent fields. You have no say over what identity class is used or what data is used to create identity values. The only requirement placed on JDO vendors implementing datastore identity is that the class they use for JDO identity objects meets the following criteria:

- The class must be public.
- The class must be serializable.
- All non-static fields of the class must be public and serializable.
- The class must have a public no-args constructor.
- The class must have a public `String` constructor. It must override the `toString` method to return a string that can be used by this constructor to create a new JDO identity object that compares equal to the instance the string was obtained from.

The last criterion listed is particularly important. As you will see in the chapter on `PersistenceManagers`, it allows you to store the identity object for a persistent instance as a string, then later recreate the identity object and retrieve the corresponding persistent instance.

4.5.2. Application Identity

Application identity is managed by you, the developer. Under application identity, the values of one or more persistent fields in an object determine its JDO identity. The fields whose values make up the object's identity are called *primary key* fields. Each object's primary key fields must be unique among all other objects of the same type.

When using application identity, it is up to you to supply the class used for JDO identity objects. This application identity class must meet all of the criteria listed for datastore identity classes. It must also obey the following requirements:

- The names of the non-static fields of the class must include the names of the primary key fields of the corresponding persistence-capable class, and the field types must be identical.
- The `equals` and `hashCode` methods of the class must use the values of all fields corresponding to primary key fields in the persistence-capable class.
- If the class is an inner class, it must be `static`.
- All classes related by inheritance must use the same application identity class.
- Each inheritance tree must use a unique application identity class.

These criteria allow you to construct an application identity object from either the values of the primary key fields of a persistent instance, or from a string produced by the `toString` method of another identity object.

Though it is not a requirement, you should also use your application identity class to register the corresponding persistence capable class with the JVM. This is typically accomplished with a static block in the application identity

class code:

```
/**
 * Application identity class for persistence capable class Magazine.
 */
public class MagazineId
{
    static
    {
        // register Magazine with the JVM
        Class c = Magazine.class;
    }

    // rest of code...
}
```

This registration process is a workaround for a quirk in JDO's persistent type registration system whereby some by-id lookups might fail if the type being looked up hasn't been used yet in your application.

Note

Though you may still create application identity classes by hand, Kodo JDO provides the `com.solarmetric.kodo.enhance.ApplicationIdTool` to automatically generate application identity classes that meet the above requirements.

4.6. Conclusions

This chapter covered everything you need to know to write persistent class definitions in JDO. JDO implementations cannot use your persistent classes, however, until you complete one additional step: you must create the JDO metadata. The next chapter explores metadata in detail.

Chapter 5. Metadata

JDO requires that you accompany each persistence-capable class with JDO metadata. This metadata serves three primary purposes:

1. To identify persistence-capable classes.
2. To override JDO default behavior.
3. To provide the JDO implementation with information that it cannot glean from simply reflecting on the persistence-capable class.

Metadata is specified as a document in the eXtensible Markup Language (XML). The Document Type Definition (DTD) for metadata documents is given in the next section. Do not worry about digesting the entire DTD immediately; we will fully cover each aspect of metadata in turn.

5.1. Metadata DTD

```
<!ELEMENT jdo (package)+>
<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>
<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|none) 'datastore'>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>
<!ELEMENT field ((collection|map|array)?, (extension)*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier (persistent|transactional|none) 'persistent'>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value (exception|default|none) 'none'>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>
<!ELEMENT array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ELEMENT collection (extension)*>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>
<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>
<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

The root element of all metadata documents is the `jdo` element. The only legal children of the `jdo` element are package elements. Each package element must specify a name attribute giving the full name of the package it represents.

Example 5.1. Basic Structure of Metadata Documents

```
<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
```

```

    ...
  </package>
  <package name="org.mag.subscribe">
    ...
  </package>
</jdo>

```

package elements contain one or more `class` elements, followed by zero or more `extension` elements. Extensions are used to annotate metadata with vendor-specific information. The `extension` element may contain other `extension` elements as children, and has three attributes:

- `vendor-name`: The name of the vendor the extension applies to. This attribute is required.
- `key`: The name of the property you are setting with the extension. Each vendor will supply a list of supported properties.
- `value`: The value of the property.

Note

Kodo JDO defines many useful metadata extensions. See the Kodo JDO Reference Guide chapter on metadata extensions for a full list.

Every persistence-capable class in the package named by each `package` element must be represented by a `class` element. Before we explore this element in detail, a brief note on how JDO resolves class names is in order.

Several metadata attributes require you to specify class names. The names you give should follow the following guidelines:

- If the class is in the package named by the current `package` element, you can give just the class name, without specifying the package. For example, if the current package name is `org.mag` and the class is `org.mag.Magazine`, then you can simply write `Magazine` for the class name.
- Similarly, if the class is in `java.lang`, `java.util`, or `java.math` packages, you do not need to specify the package in the class name.
- Otherwise, the full class name is required, including package name.
- If the class is an inner class, then write it as `parent-class$inner-class`. For example, `SubscriptionForm$LineItem`.

We now turn our attention back to the `class` element. This element has the following attributes:

- `name`: The name of the class. This attribute is required.
- `persistence-capable-superclass`: If the superclass of this class is also persistent, and you wish JDO to know about the inheritance structure, then you must name the superclass in this attribute. If the superclass of this class is not persistent or if for some reason you want JDO to treat the superclass as unrelated, you should not specify this attribute.
- `identity-type`: Gives the JDO identity type used by the class. Legal values are `application` for application identity, `datastore` for datastore identity, and `none`. This attribute defaults to a value of `application` if the `objectid-class` attribute is specified, and `datastore` otherwise.
- `objectid-class`: For application identity, the name of the JDO identity class used by this class. The `objectid-class` should only be given for the base class in the inheritance tree; subclasses will "inherit" the

identity class from their superclass.

- `requires-extent`: Set this attribute to `false` if you will never need to query for persistent instances of this class (i.e., if all objects of the class can be obtained through JDO identity lookups or through relations with other objects). Defaults to `true`.

Example 5.2. Metadata Class Listings

```
<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      ...
    </class>
    <class name="Article">
      ...
    </class>
    <class name="Author">
      ...
    </class>
    <class name="Address">
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Form">
      ...
    </class>
    <class name="SubscriptionForm" persistence-capable-superclass="Form">
      ...
    </class>
    <class name="SubscriptionForm$LineItem">
      ...
    </class>
  </package>
</jdo>
```

The `class` element may contain extension elements and `field` elements. `field` elements represent fields declared by the persistence-capable class. These elements are optional; if a field declared in the class is not named by some `field` element, then its properties are defaulted as explained in the attribute listings below. Thanks to JDO's comprehensive set of defaults, most fields do not need to be listed explicitly. `field` elements may have the following attributes:

- `name`: The name of the field, as it is declared in the persistence-capable class. This attribute is required.
- `persistence-modifier`: Specifies how JDO should manage the field. Legal values are `persistent` for persistent fields, `transactional` for fields that are non-persistent but can be rolled back along with the current transaction, and `none`. The default value of this attribute is based on the type of the field:
 - Fields declared `static`, `transient`, or `final` default to `none`.
 - Fields of any primitive or primitive wrapper type default to `persistent`.
 - Fields of types `java.lang.String`, `java.lang.Number`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Locale`, and `java.util.Date` default to `persistent`.
 - Fields of any user-defined persistence-capable type default to `persistent`.

- Arrays of any of the types mentioned so far default to `persistent`.
- Fields of the following container types in the `java.util` package default to `persistent`: `Collection`, `Set`, `List`, `Map`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, `Vector`.
- All other fields default to `none`.
- `primary-key`: Set this attribute to `true` if the class uses application identity and this field is a primary key field. Defaults to `false`.
- `null-value`: Specifies the treatment of null values when the field is written to the data store. Use a value of `none` if the data store should hold a null value for the field. Use `default` to write a data store default value instead. Finally, use `exception` if you want the JDO implementation to throw an exception if the field contains a null value when it is being written to the data store. Defaults to `none`.
- `default-fetch-group`: Default fetch group fields are managed together as a group for efficiency. They are typically loaded as a block from the data store, and are often written as a block as well. This attribute defaults to `true` for primitive, primitive wrapper, `String`, `Date`, `BigDecimal`, and `BigInteger` types. All other types default to `false`.
- `embedded`: This is a hint to the JDO implementation to store the field as part of the class instance in the data store, rather than as a separate entity. JDO implementations are free to ignore this attribute. Its value defaults to `true` for primitive, primitive wrapper, `Date`, `BigDecimal`, `BigInteger`, and array types. All other types default to `false`.

All field elements may contain extension child elements. field elements that represent array, collection, or map fields may also contain a single array, collection, or map child element, respectively. Each of these elements may contain additional extension elements in turn.

The array element has a single attribute, `embedded-element`. This attribute mirrors the `embedded` attribute of the class element, but applies to the values stored in each array index.

The collection element also has the `embedded-element` attribute. Additionally, it declares the `element-type` attribute. Use this attribute to tell the JDO implementation what class of objects the collection contains. If the `element-type` is not given, it defaults to `java.lang.Object`.

map elements define four attributes. They are:

- `key-type`: The class of objects used for map keys. Defaults to `java.lang.Object`.
- `embedded-key`: Same as the `embedded-element` element of arrays and collections, but applies to map keys.
- `value-type`: The class of objects used for map values. Defaults to `java.lang.Object`.
- `embedded-value`: Same as the `embedded-element` element of arrays and collections, but applies to map values.

That exhausts the metadata document structure. A complete metadata document example is presented below.

Example 5.3. Complete Metadata Document

```
<?xml version="1.0"?>
<!-- Note that all persistence-capable classes must be listed, but -->
```

```

<!-- very few fields need to be specified -->
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      <field name="isbn" primary-key="true"/>
      <field name="articles">
        <collection element-type="Article"/>
      </field>
    </class>
    <class name="Article">
      <field name="authors">
        <map key-type="String" value-type="Author"/>
      </field>
    </class>
    <class name="Author"/>
    <class name="Address"/>
  </package>
  <package name="org.mag.subscribe">
    <class name="Form"/>
    <class name="SubscriptionForm" persistence-capable-superclass="Form">
      <field name="lineItems">
        <collection element-type="SubscriptionForm$LineItem">
          <extension vendor-name="kodo" key="inverse" value="form"/>
        </collection>
      </field>
    </class>
    <class name="SubscriptionForm$LineItem"/>
  </package>
</jdo>

```

5.2. Metadata Placement

JDO metadata must be available both during class enhancement and at runtime. The metadata document listing a persistence-capable class must be available as a resource from the class' class loader, and must exist in one of two standard locations:

1. In a resource called `class-name.jdo`, where `class-name` is the name of the class the document applies to, without package name. The resource must be located in the same package as the class.
2. In a resource called `package-name.jdo`, where `package-name` is the last token of the package's full name. The resource should be placed in the corresponding package, or in the package's parent package. Package-level documents should contain the metadata for all the persistence-capable classes in the package, except those classes that have individual `class-name.jdo` resources associated with them.

Assuming you are using a standard Java class loader, these rules imply that for a class `Magazine` defined by the file `org/mag/Magazine.class`, the corresponding metadata document could be defined in any of the following files:

- `org/mag/Magazine.jdo`
- `org/mag/mag.jdo`
- `org/mag.jdo`

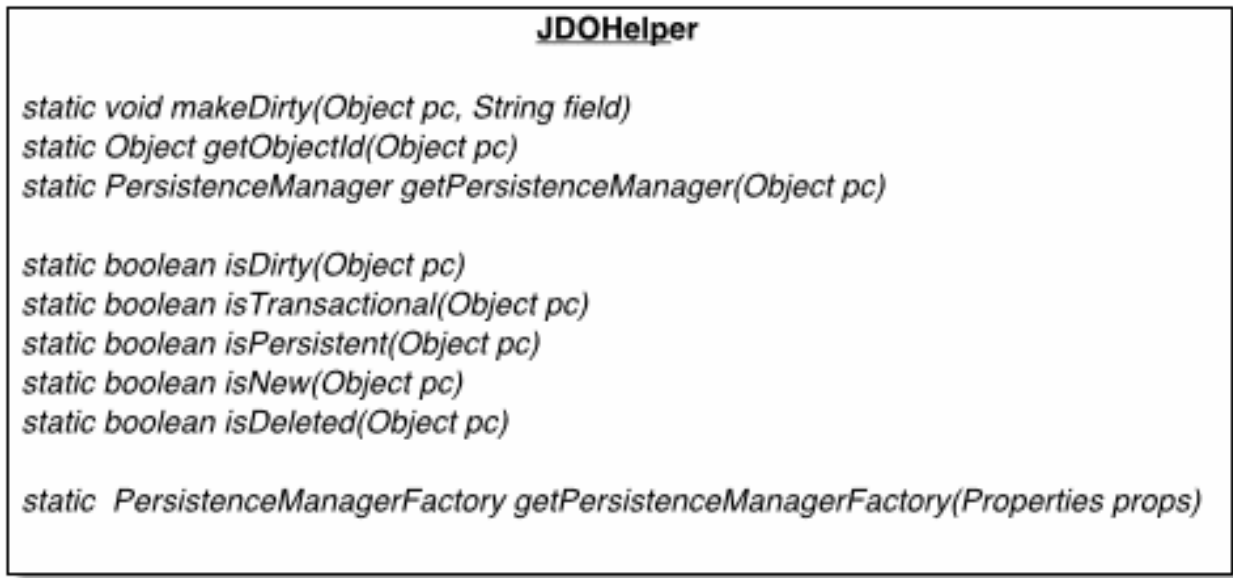
Because metadata documents are loaded as resources, JDO implementations can also read them from `jar` files.

Note

Some JDO implementations may not support all of the metadata placements discussed, or may support additional placements. Kodo JDO supports the standard metadata locations listed as well as a package-level resource called `package.jdo` and a `system.jdo` resource defining the metadata for the entire system,

located in any top-level directory of the CLASSPATH.

Chapter 6. JDOHelper



The above diagram depicts the most commonly-used methods of the `javax.jdo.JDOHelper` class. For a complete API reference, consult the class Javadoc.

Applications use the `JDOHelper` for three types of operations: persistence-capable operations, lifecycle operations, and `PersistenceManagerFactory` construction. We investigate each below.

6.1. Persistence-Capable Operations

We have already seen the first two persistence-capable operations, `makeDirty` and `getObjectId`. Given a persistence-capable object and the name of the field that has been modified, the `makeDirty` method notifies the JDO implementation that the field's value has changed so that it can write the new value to the data store. JDO usually tracks field modifications automatically; the only time you are required to use this method is when you assign a new value to some index of a persistent array.

The `getObjectId` method returns the JDO identity object for the persistence-capable instance given as an argument. If the given instance is not persistent, this method returns `null`.

The final persistence-capable operation, `getPersistenceManager`, is self-explanatory. It simply returns the `PersistenceManager` that is managing the persistence-capable object supplied as an argument. If the argument is a *transient* object, meaning it is not managed by a `PersistenceManager`, `null` is returned.

6.2. Lifecycle Operations

JDO recognizes several lifecycle states for persistence-capable objects. Instances transition between these states according to strict rules defined in the JDO specification. State transitions can be triggered by both explicit actions, such as calling the `deletePersistent` method of a `PersistenceManager` to delete a persistent object, and by implicit actions, such as reading or writing a persistent field.

The list below enumerates the lifecycle states for persistence-capable instances. Unless otherwise noted, each state must be supported by all JDO implementations. Do not concern yourself with memorizing the states and transitions presented; you will rarely need to think about them in practice.

Note

Some of the state transitions mentioned below occur at transaction boundaries. If you are unfamiliar with transactions, you may want to read the first few paragraphs of the chapter on the `Transaction` interface to become familiar with the concepts involved before continuing.

- *Transient*. Objects that are created via a user-defined constructor and have no association with the persistence framework are called transient objects. Transient objects behave exactly as if JDO does not exist.

A transient object transitions to persistent-new if it is the parameter of a call to the `PersistenceManager`'s `makePersistent` method.

- *Persistent-new*. The persistent-new state is reserved for objects that have been made persistent by a call to `makePersistent`, but have not yet been inserted into the data store. When an object transitions to the persistent-new state, it is given a JDO identity.

A persistent-new instance transitions to persistent-new-deleted if it is an argument to the `PersistenceManager`'s `deletePersistent` method.

On transaction commit, the information in the persistent-new object is inserted into the data store. The object transitions to persistent-nontransactional if the `Transaction`'s `RetainValues` property is set to `true`, otherwise it transitions to hollow. On transaction rollback, a persistent-new instance returns to the transient state. The data store is not affected. If the `Transaction`'s `RestoreValues` property is set to `true`, the instance's persistent and transactional fields will be restored to the values they had when the transaction began.

- *Persistent-new-deleted*. Objects that have been both persisted with `makePersistent` and then deleted with `deletePersistent` in the current transaction wind up in the persistent-new-deleted state. When objects are in this state, you are only allowed to access their primary key fields. Attempting to access any other persistent field will result in a `JDOUserException`.

A persistent-new-deleted object transitions to transient on transaction commit. The values of its persistent fields are replaced with Java default values. A persistent-new-deleted object also becomes transient if the transaction is rolled back. In this case, its persistent and transactional fields will be restored to the values they had when the transaction began if the `Transaction`'s `RestoreValues` property is `true`, else they will be left untouched.

- *Persistent-clean*. Objects that represent specific state in the data store and whose persistent fields have not been changed in the current transaction are persistent-clean.

A persistent-clean instance transitions to persistent-dirty if a change is made to any of its persistent fields. It transitions to persistent-deleted if it is the parameter of a call to the `PersistenceManager`'s `deletePersistent` method. It transitions to persistent-nontransactional if it is the parameter of a call to `makeNontransactional`. It transitions to transient if it is the parameter of a call to `makeTransient`.

With the `Transaction`'s `RetainValues` property set to `true`, a persistent-clean object transitions to persistent-nontransactional on transaction commit, otherwise it transitions to hollow. Correspondingly, with `RestoreValues` set to `true`, a persistent-clean object transitions to persistent-nontransactional on transaction rollback, otherwise it transitions to hollow.

- *Persistent-dirty*. Persistent objects that have been changed within the current transaction are persistent-dirty.

A persistent-dirty object transitions to persistent-deleted if it is the parameter of a call to `PersistenceManager.deletePersistent`.

On transaction commit, the data store record the persistent-dirty instance represents will be updated to reflect the values in the object's persistent fields. The lifecycle state transitions for a persistent-dirty instance on transaction

completion are the same as those outlined above for a persistent-clean instance.

- *Persistent-deleted.* If a persistent object is the parameter of a call to the `PersistenceManager`'s `deletePersistent` method, it becomes persistent-deleted. When an object is in this state, you are only allowed to access its primary key fields. Attempting to access any other persistent field will result in a `JDOUserException`.

A persistent-deleted object transitions to transient on transaction commit. The data store record for the object is removed. The values of the instance's persistent fields are replaced with Java default values. On transaction rollback, a persistent-deleted object transitions to persistent-nontransactional if the `Transaction`'s `RetainValues` property is set to `true`, else it transitions to hollow.

- *Hollow.* Persistent objects whose values have not been loaded from the data store are in the hollow state. Whenever an instance transitions to hollow, its persistent fields are cleared and replaced with their Java default values. The fields will be re-loaded with their data store values the first time you access them. Delaying the loading of persistent information until it is needed is known as *lazy loading*.

JDO implementations use only weak or soft references to track hollow instances, so they may be garbage collected if your application does not hold strong references to them.

If you are using optimistic transactions, a hollow instance will transition to persistent-nontransactional if any of its persistent, non-primary-key fields are read. Under data store transactions, the instance will transition to persistent-clean. Writing to a persistent field of a hollow instance will cause it to transition to persistent-dirty, regardless of transaction type.

- *Persistent-nontransactional.* Persistent-nontransactional objects represent persistent data in the data store, but are not guaranteed to reflect the most current values of that data. A lifecycle state that allows access to data that may be outdated might sound useless; if they are utilized carefully, however, persistent-nontransactional objects can sometimes offer large performance gains, with little danger of employing stale data in your application.

The persistent-nontransactional state is an optional feature of JDO, and may not be supported in many implementations. It is also by far the most complex lifecycle state. It is governed by the `NontransactionalRead`, `NontransactionalWrite`, `RetainValues`, and `Optimistic` properties of the `Transaction`. Implementations may support any or all of these properties. These properties are detailed in the section explaining `PersistenceManagerFactory` properties.

Outside of a transaction, reading and writing persistent fields of a persistent-nontransactional instance does not result in any state change. Any modifications you make to the instance's persistent fields will be discarded the next time it enters a data store transaction. Within this type of transaction, reading a persistent field of a persistent-nontransactional instance causes a transition to persistent-clean, and writing a persistent field causes a transition to persistent-dirty. Within an optimistic transaction, reading a persistent field of a persistent-nontransactional instance does not change the instance's state; writing a persistent field causes a transition to persistent-dirty.

A persistent-nontransactional object transitions to persistent-clean if it is the parameter of a call to the `PersistenceManager`'s `makeTransactional` method. It transitions to persistent-deleted if it is the parameter of a call to `makeDeleted`. It transitions to transient if it is passed to the `PersistenceManager`'s `makeTransient` method.

- *Transient-clean.* The transient-clean and transient-dirty states are grouped together in the *transient-transactional* lifecycle category. Transient-transactional objects are not persistent, but their fields recognize transaction boundaries, meaning they can be restored to their previous values when a transaction is rolled back. You can make a transient instance transient-transactional by passing it to the `PersistenceManager`'s `makeTransactional` method. Some JDO vendors may not support the transient-transactional states; they are an optional feature of the JDO specification.

Transient-transactional objects that have not been changed in the current transaction are transient-clean. A

transient-clean object transitions to transient-dirty if any of its persistent or transactional fields are changed within a transaction. It transitions to transient if it is the parameter of a call to `PersistenceManager.makeNontransactional`.

- *Transient-dirty*. Transient-transactional instances that have been modified in the current transaction are transient-dirty. On transaction completion, a transient-dirty object transitions to transient-clean. If the Transaction is rolled back and its `RestoreValues` property is `true`, the persistent and transactional fields of a transient-dirty object will be restored to the values they had when the transaction began.

Note

Kodo JDO supports all JDO lifecycle states, including all optional states.

After reviewing the JDO lifecycle states, the purpose of the JDOHelper's lifecycle operations -- `isDirty`, `isTransactional`, `isPersistent`, `isNew`, `isDeleted` -- should be clear. Each one tells you whether or not the given persistence-capable instance has a certain property (`dirty`, `transactional`, `persistent`, `new`, and `deleted`, respectively), where these properties are determined by the lifecycle state of the instance. In fact, you can calculate the exact state of the instance based on these properties according to the table below. Once again, however, you will rarely worry about the lifecycle state of your persistence-capable objects in practice.

| | Persistent | Transactional | Dirty | New | Deleted |
|-----------------------------|------------|---------------|-------|-----|---------|
| Transient | | | | | |
| Transient-clean | | ✓ | | | |
| Transient-dirty | | ✓ | ✓ | | |
| Persistent-new | ✓ | ✓ | ✓ | ✓ | |
| Persistent-nontransactional | ✓ | | | | |
| Persistent-clean | ✓ | ✓ | | | |
| Persistent-dirty | ✓ | ✓ | ✓ | | |
| Hollow | ✓ | | | | |
| Persistent-deleted | ✓ | ✓ | ✓ | | ✓ |
| Persistent-new-deleted | ✓ | ✓ | ✓ | ✓ | ✓ |

6.3. PersistenceManagerFactory Construction

You can use the `getPersistenceManagerFactory` method of the JDOHelper to obtain `PersistenceManagerFactory` objects in a vendor-neutral fashion. This method takes a single argument, a

`java.util.Properties` instance. The `Properties` instance is used to configure the `PersistenceManagerFactory` before it is returned from the method. Vendors may construct a new `PersistenceManagerFactory` with each invocation of this method, or may pool `PersistenceManagerFactory` instances and return a pooled instance that matches the supplied properties. The available configuration options and their associated property names are discussed in the next chapter detailing the `PersistenceManagerFactory` interface.

Example 6.1. Obtaining a `PersistenceManagerFactory`

```
// this is usually just done once in your application somewhere, and then
// you cache the factory for easy retrieval by application components; often
// the properties are read from a properties file
Properties props = new Properties ();

// this property key tells the jdohelper what pmfactory class to instantiate
props.setProperty ("javax.jdo.PersistenceManagerFactoryClass",
    "com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory");

// these properties define the default settings for persistence managers
// produced by this factory; these settings are covered in the next chapter
props.setProperty ("javax.jdo.option.Optimistic", "true");
props.setProperty ("javax.jdo.option.RetainValues", "true");
props.setProperty ("javax.jdo.option.ConnectionUserName", "solarmetric");
props.setProperty ("javax.jdo.option.ConnectionPassword", "kodo");
props.setProperty ("javax.jdo.option.ConnectionURL", "jdbc:hsql:database");
props.setProperty ("javax.jdo.option.ConnectionDriverName",
    "org.hsqldb.jdbcDriver");

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory (props);
```

Chapter 7. PersistenceManagerFactory

```

PersistenceManagerFactory

String ConnectionUserName
String ConnectionPassword
String ConnectionURL
String ConnectionDriverName
String ConnectionFactoryName
String ConnectionFactory2Name
Object ConnectionFactory
Object ConnectionFactory2

boolean Multithreaded
boolean Optimistic
boolean RetainValues
boolean RestoreValues
boolean NontransactionalRead
boolean NontransactionalWrite
boolean IgnoreCache

int MaxPool
int MinPool
int MsWait

PersistenceManager getPersistenceManager()
PersistenceManager getPersistenceManager(String user, String pass)

Collection supportedOptions()
Properties getProperties()

```

The `PersistenceManagerFactory` creates `PersistenceManager` instances for application use. It allows you to configure data store connectivity and to specify the default settings of the `PersistenceManagers` it constructs. You can also use it to programmatically discover what JDO options your current vendor supports, enabling you to build applications that optimize themselves for full-featured products, but still function under more basic JDO implementations.

7.1. Obtaining a PersistenceManagerFactory

JDO vendors may supply public constructors for their `PersistenceManagerFactory` implementations, but the recommended method of obtaining a `PersistenceManagerFactory` is through the `JDOHelper`'s `getPersistenceManagerFactory` method. This method's `Properties` parameter supplies the configuration for the factory. `PersistenceManagerFactory` objects returned from the `getPersistenceManagerFactory` method are "frozen"; any attempt to change their property settings will result in a `JDOUserException`. This is because the returned factory may come from a pool, and might be shared

by other application components.

JDO requires that concrete `PersistenceManagerFactory` classes implement the `Serializable` interface. This allows you to create and configure a `PersistenceManagerFactory`, then serialize it to a file or store it in a Java Naming and Directory Interface (JNDI) tree for later retrieval and use.

7.2. PersistenceManagerFactory Properties

The majority of the `PersistenceManagerFactory` interface consists of Java bean-style "getter" and "setter" methods for several properties, represented by field declarations in the diagram at the beginning of this chapter. These properties can be grouped into three functional categories: data store connection configuration, default `PersistenceManager` and Transaction options, and `PersistenceManager` pooling settings.

The lists below explain the meaning of each property. Many of these properties can be set through the `Properties` instance passed to the aforementioned `getPersistenceManagerFactory` method in `JDOHelper`. Where this is the case, the name of the corresponding `Properties` key will appear in parentheses after the property name.

7.2.1. Connection Configuration

Use the properties below to tell JDO implementations how to connect with your data store.

- *ConnectionUserName* (`javax.jdo.option.ConnectionUserName`). The user name to specify when connecting to the data store.
- *ConnectionPassword* (`javax.jdo.option.ConnectionPassword`). The password for the above user.
- *ConnectionURL* (`javax.jdo.option.ConnectionURL`). The URL of the data store.
- *ConnectionDriverName* (`javax.jdo.option.ConnectionDriverName`). The full class name of the driver to use when interacting with the data store.
- *ConnectionFactoryName* (`javax.jdo.option.ConnectionFactoryName`). The JNDI location of a connection factory to use to obtain data store connections. This property overrides the other data store connection properties above.
- *ConnectionFactory*. A connection factory to use to obtain data store connections. This property overrides all other data store connection properties above, including the *ConnectionFactoryName*. The exact type of the given factory is implementation-dependent. Many JDO implementations will expect a standard Java Connector Architecture (JCA) *ConnectionFactory*. Implementations layered on top of JDBC might expect a JDBC *DataSource*. Still other implementations might use other factory types.
- *ConnectionFactory2Name* (`javax.jdo.option.ConnectionFactory2Name`). In a managed environment, connections obtained from the primary connection factory may be automatically enlisted in any global transaction in progress. To implement local transactions, the JDO implementation might require an additional connection factory that is not configured to participate in global transactions. The JNDI location of this factory can be specified through this property.
- *ConnectionFactory2*. The connection factory to use for local transactions. Overrides the *ConnectionFactory2Name* above.

Note

Kodo JDO uses JDBC *DataSources* in place of JCA *ConnectionFactory* objects.

7.2.2. PersistenceManagerFactory and Transaction Defaults

The settings below will become the default property values for the `PersistenceManagers` and associated `Transactions` produced by the `PersistenceManagerFactory`. Some implementations may not fully support all properties. If you attempt to set a property to an unsupported value, the operation will throw a `JDOUnsupportedOptionException`.

- *Multithreaded* (`javax.jdo.option.Multithreaded`). Set this property to `true` to indicate that the persistence-capable instances managed by the `PersistenceManager` will be accessed concurrently by multiple threads.
- *Optimistic* (`javax.jdo.option.Optimistic`). Set to `true` to use optimistic transactions by default.
- *RetainValues* (`javax.jdo.option.RetainValues`). If this property is `true`, the fields of persistent objects will not be cleared on transaction commit. `RetainValues` automatically implies `NontransactionalRead`.
- *RestoreValues* (`javax.jdo.option.RestoreValues`). Controls the behavior of persistent and transactional fields on transaction rollback. Set this property to `true` to restore the fields to the values they had when the transaction began.
- *NontransactionalRead* (`javax.jdo.option.NontransactionalRead`). Specifies whether you can read persistent fields outside of a transaction. If this property is `false`, any attempt to read a persistent field outside of a transaction will result in a `JDOUserException`.
- *NontransactionalWrite* (`javax.jdo.option.NontransactionalWrite`). Specifies whether you can write to persistent fields outside of a transaction. If this property is `false`, any attempt to modify a persistent field outside of a transaction will result in a `JDOUserException`.
- *IgnoreCache* (`javax.jdo.option.IgnoreCache`). This property controls whether changes made to persistent instances in the current transaction are considered when evaluating queries. A value of `true` is a hint to the JDO runtime that changes in the current transaction can be ignored; this usually enables the implementation to run the query using the data store's native query interface. A value of `false`, on the other hand, may force implementations to run transactional queries in memory, which is often a slow process.

Note

Kodo JDO supports all `PersistenceManager` and `Transaction` properties. It recognizes many additional properties as well; see the Kodo JDO Reference Guide for details.

7.2.3. PersistenceManager Pooling

The following properties apply to the `PersistenceManagerFactory`'s pooling of `PersistenceManagers`. Implementations are free to ignore these property settings if they do not support `PersistenceManager` pooling.

- *MaxPool* (`javax.jdo.option.MaxPool`). The maximum number of `PersistenceManagers` to allow in the pool.
- *MinPool* (`javax.jdo.option.MinPool`). The minimum number of `PersistenceManagers` to allow in the pool.
- *MsWait* (`javax.jdo.option.MsWait`). The number of milliseconds to wait for a free `PersistenceManager` before giving up.

Note

Kodo JDO applies these settings to its automatic connection pooling; it does not pool `PersistenceManagers`.

7.3. Obtaining PersistenceManagers

The `PersistenceManagerFactory` interface includes two `getPersistenceManager` methods for obtaining `PersistenceManager` instances. One version takes as parameters the user name and password to use for the `PersistenceManager`'s data store connection(s). The other version relies on the `ConnectionUserName` and `ConnectionPassword` settings of the `PersistenceManagerFactory`. Both methods may return a newly-constructed `PersistenceManager`, or may return one from a pool of instances.

After the first `PersistenceManager` is acquired from a `PersistenceManagerFactory`, the factory's configuration is "frozen". Any attempt to change its properties will result in a `JDOUserException`.

7.4. Properties and Supported Options

In addition to supplying `PersistenceManagers`, the `PersistenceManagerFactory` also supplies metadata about the current JDO implementation. The `getProperties` method returns a `Properties` instance containing, at a minimum, the following keys:

- `VendorName`: The name of the JDO vendor.
- `VersionNumber`: The version number string for the product.

The `supportedOptions` method returns a `Collection` of `Strings` enumerating the JDO options supported by the implementation. The following option names are recognized:

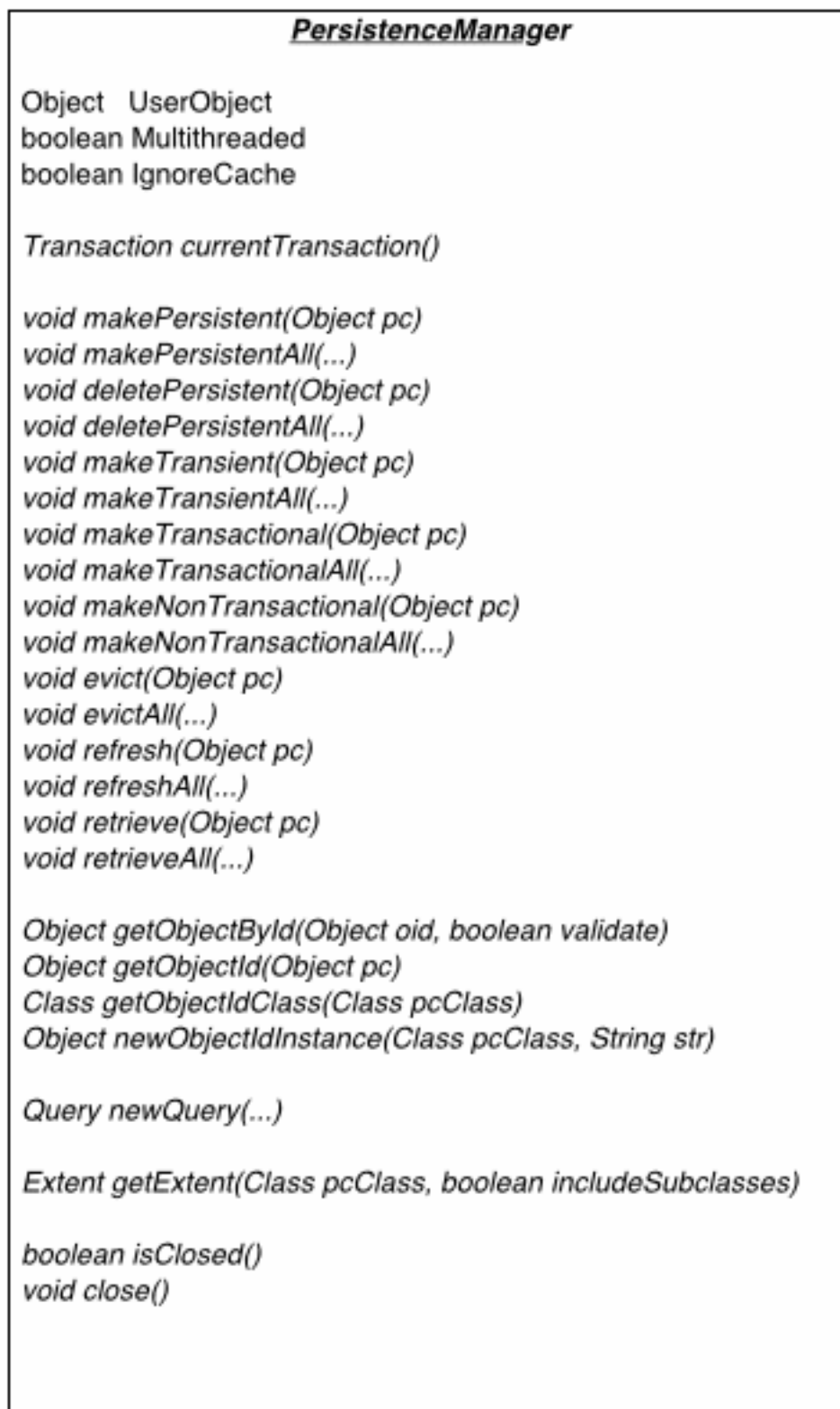
- `javax.jdo.option.TransientTransactional`
- `javax.jdo.option.NontransactionalRead`
- `javax.jdo.option.NontransactionalWrite`
- `javax.jdo.option.RetainValues`
- `javax.jdo.option.Optimistic`
- `javax.jdo.option.ApplicationIdentity`
- `javax.jdo.option.DatastoreIdentity`
- `javax.jdo.option.NondurableIdentity`
- `javax.jdo.option.ArrayList`
- `javax.jdo.option.HashMap`
- `javax.jdo.option.Hashtable`
- `javax.jdo.option.LinkedList`

- `javax.jdo.option.TreeMap`
 - `javax.jdo.option.TreeSet`
 - `javax.jdo.option.Vector`
 - `javax.jdo.option.Map`
 - `javax.jdo.option.List`
 - `javax.jdo.option.Array`
 - `javax.jdo.option.NullCollection`
 - `javax.jdo.option.ChangeApplicationIdentity`
 - `javax.jdo.query.JDOQL`
- Vendors may include Strings for other query languages they support as well.

Note

Kodo JDO currently supports all options except `javax.jdo.option.NullCollection` and `javax.jdo.option.ChangeApplicationIdentity`.

Chapter 8. PersistenceManager



The diagram above presents an overview of the most commonly-used methods and properties of the

PersistenceManager interface. For a complete treatment of the PersistenceManager API, see the Javadoc documentation. Java bean-like properties with "getter" and "setter" methods are listed as field declarations. Methods whose parameter signatures consist of an ellipses (...) are overloaded to take multiple parameter types.

The PersistenceManager is the primary interface used by application developers to interact with the JDO runtime. Each PersistenceManager manages a cache of persistent and transactional objects, and has an association with a single Transaction.

The methods of the PersistenceManager can be divided into the following functional categories:

- User object association.
- Configuration properties.
- Transaction association.
- Persistence-capable lifecycle management.
- JDO identity management.
- Query factory.
- Extent factory.
- Closing.

8.1. User Object Association

The PersistenceManager's UserObject property allows you to associate an arbitrary object with each PersistenceManager. The given object is not used in any way by the JDO implementation.

8.2. Configuration Properties

The PersistenceManager interface includes "getter" and "setter" methods for two configuration properties: Multithreaded and IgnoreCache. These properties are discussed in the section detailing the PersistenceManagerFactory settings.

8.3. Transaction Association

Every PersistenceManager has a one-to-one relation with a Transaction instance; in fact, many vendors use a single class to implement both the PersistenceManager and Transaction interfaces. If your application requires multiple concurrent transactions, you will have to use multiple PersistenceManagers.

You can retrieve the Transaction associated with a PersistenceManager through the currentTransaction method.

8.4. Persistence-capable Lifecycle Management

PersistenceManagers perform several actions that affect the lifecycle state of persistence-capable instances. Each of these actions is represented by multiple methods in the PersistenceManager interface: one method that acts on a single persistence-capable object, such as makePersistent, and corresponding methods that accept a collection or array of persistence-capable objects, such as makePersistentAll.

- `makePersistent(All)`: Transitions transient instances to persistent-new. This action can only be used in the context of an active transaction. When the transaction is committed, the newly persisted instances will be inserted into the data store.
- `deletePersistent(All)`: Transitions persistent instances to persistent-deleted, or persistent-new instances to persistent-new-deleted. This action, too, can only be called during an active transaction. The given instance(s) will be deleted from the data store when the transaction is committed.
- `makeTransient(All)`: This action transitions persistent instances to transient. The instances immediately lose their association with the `PersistenceManager` and their JDO identity. The data store records for the instances are not modified.

This action can only be run on clean objects. If it is run on a dirty object, a `JDOUserException` is thrown.

- `makeTransactional(All)`: Use this action to make transient instances transient-transactional, or to bring persistent-nontransactional instances into the current transaction. In the latter case, the action must be invoked during an active transaction.
- `makeNontransactional(All)`: Transitions transient-clean instances to transient, and persistent-clean instances to persistent-nontransactional. Invoking this action on a dirty instance will result in a `JDOUserException`.
- `evict(All)`: Evicting an object tells the `PersistenceManager` that your application no longer needs that object. The object transitions to hollow and the `PersistenceManager` releases all strong references to it, allowing it to be garbage collected.

Calling the `evictAll` method with no parameters acts on all persistent-clean objects in the `PersistenceManager`'s cache.

- `refresh(All)`: Use the `refresh` action to make sure the persistent state of an instance is in synch with the values in the data store. `refresh` is intended for long-running optimistic transactions in which there is a danger of seeing stale data.

Calling the `refreshAll` method with no parameters acts on all transactional objects in the cache. If there is no transaction in progress, the method is a no-op.

- `retrieve(All)`: Retrieving a persistent object immediately loads all of the object's persistent fields with their data store values. You might use this action to make sure an instance's fields are fully loaded before transitioning it to transient.

8.5. JDO Identity Management

Each `PersistenceManager` is responsible for managing the JDO identities of the persistent objects in its cache. The following methods allow you to interact with the management of JDO identities:

- `getObjectIdClass`: Returns the JDO identity class used for the given persistence-capable class.
- `newObjectIdInstance`: This method is used to re-create JDO identity objects from the string returned by their `toString` method. Given a persistence-capable class and a JDO identity string, the method constructs a JDO identity object. Using the `getObjectById` method described below, this identity object can then be employed to obtain the persistent instance whose identity was used to create the string in the first place.
- `getObjectId`: Returns the JDO identity object for a persistent instance managed by this `PersistenceManager`.
- `getObjectById`: This method returns the persistent instance corresponding to the given JDO identity object.

If the instance is already cached, the cached version will be returned. Otherwise, a new instance will be constructed, and may or may not be loaded with data from the data store (some implementations might return a hollow instance).

If the `validate` parameter of this method is set to `true`, then the JDO implementation will throw an exception if the data store record for the given JDO identity does not exist. Otherwise, some implementations might return a hollow instance for the missing record, and an exception will not be thrown until you attempt to access one of its persistent fields.

8.6. Extent Factory

Extents are logical representations of all persistent instances of a given persistence-capable class, possibly including subclasses.

Extents are obtained through the `PersistenceManager`'s `getExtent` method. This method takes two parameters: the class of objects the Extent contains, and a `boolean` indicating whether or not subclasses are included as well.

You can only retrieve Extents for persistence-capable classes whose metadata specifies a value of `true` for the `requires-extent` attribute (this is the default).

8.7. Query Factory

Query objects are used to find persistent objects matching certain criteria. You can obtain a `Query` through one of the `PersistenceManager`'s several `newQuery` methods. See the chapter covering the `Query` interface and the `PersistenceManager` Javadoc for details.

8.8. Closing

When a `PersistenceManager` is no longer needed, you should call its `close` method. Closing a `PersistenceManager` releases any resources it is using. The persistent instances managed by the `PersistenceManager` become invalid, as do any `Query` and `Extent` objects it created. Calling any method other than `isClosed` on a closed `PersistenceManager` results in a `JDOUserException`.

Chapter 9. Transaction

Transactions are critical to maintaining data integrity. They are used to group operations into units of work that act in an all-or-nothing fashion. Transactions have the following qualities:

- *Atomicity*. Atomicity refers to the all-or-nothing property of transactions. Either every data update in the transaction completes successfully, or they all fail, leaving the data store in its original state. A transaction cannot be only partially successful.
- *Consistency*. Each transaction takes the data store from one consistent state to another consistent state.
- *Isolation*. Transactions are isolated from each other. When you are reading persistent data in one transaction, you cannot "see" the changes that are being made to that data in other uncompleted transactions. Similarly, the updates you make in one transaction cannot conflict with updates made in other concurrent transactions. The form of conflict resolution employed depends on whether you are using pessimistic or optimistic transactions. Both types are described later in this chapter.
- *Durability*. The effects of successful transactions are durable; the updates made to persistent data last for the lifetime of the data store.

Together, these qualities are called the ACID properties of transactions. To understand why these properties are so important to maintaining data integrity, consider the following example:

Suppose you create an application to manage bank accounts. The application includes a method to transfer funds from one user to another, and it looks something like this:

```
public void transferFunds (User from, User to, double amnt)
{
    from.decrementAccount (amnt);
    to.incrementAccount (amnt);
}
```

Now suppose that user Alice wants to transfer 100 dollars to user Bob. No problem; you simply invoke your `transferFunds` method, supplying Alice in the `from` parameter, Bob in the `to` parameter, and `100.00` as the `amnt`. The first line of the method is executed, and 100 dollars is subtracted from Alice's account. But then, something goes wrong. An unexpected exception occurs, or the hardware fails, and your method never completes.

You are left with a situation in which the 100 dollars has simply disappeared. Thanks to the first line of your method, it is no longer in Alice's account, and yet it was never transferred to Bob's account either. The data store is in an inconsistent state.

The importance of transactions should now be clear. If the two lines of the `transferFunds` method had been placed together in a transaction, it would be impossible for only the first line to succeed -- either the funds would be transferred properly or they would not be transferred at all and an exception would be thrown. Money could never vanish into thin air; the data store could never get into an inconsistent state.

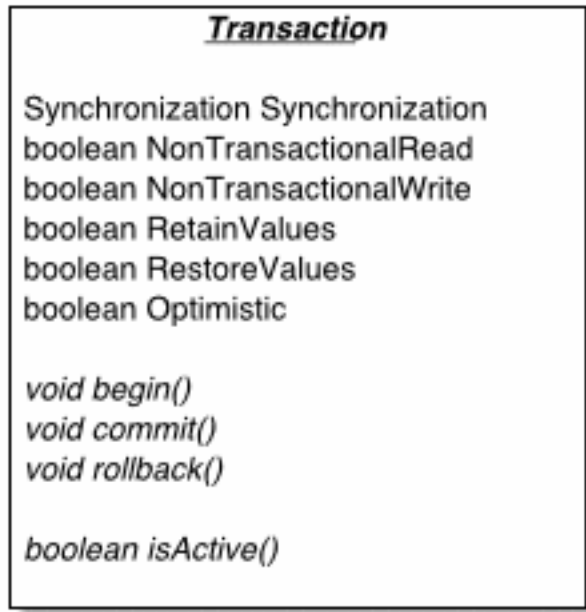
9.1. Transaction Types

There are two major types of transactions: data store, or pessimistic, transactions and optimistic transactions. Each type has both advantages and disadvantages.

Pessimistic transactions generally lock the data store records they act on, preventing other concurrent transactions from using the same data. This avoids conflicts between transactions, but consumes a lot of database resources. Additionally, locking records can result in deadlock, a situation in which two transactions are both waiting for the other to release its locks before completing. The results of a deadlock are data store-dependent; usually one transaction is forcefully rolled back after some specified time out interval, and an exception is thrown.

Optimistic transactions consume less resources than pessimistic transactions, but only at the expense of reliability. Because optimistic transactions do not lock data store records, two transactions might change the same persistent information at the same time, and the conflict will not be detected until the second transaction attempts to commit. At this time, the second transaction will realize that another transaction has concurrently modified the same records (usually through a timestamp or versioning system), and will throw an appropriate exception. Note that optimistic transactions still maintain data integrity; they are simply more likely to fail in heavily concurrent situations.

9.2. The JDO Transaction Interface



The `Transaction` interface controls transactions in JDO. This interface consists of "getter" and "setter" methods for several Java bean-style properties, standard transaction demarcation methods, and a method to test whether there is a transaction in progress.

The `Transaction`'s `NonTransactionalRead`, `NonTransactionalWrite`, `RetainValues`, `RestoreValues`, and `Optimistic` properties mirror those presented in the section on `PersistenceManagerFactory` settings. The final `Transaction` property, `Synchronization` has not been covered yet. This property enables you to associate a `javax.transaction.Synchronization` instance with the `Transaction`. The `Transaction` will notify your `Synchronization` instance on transaction completion events, so that you can implement custom behavior on commit or rollback. See the `javax.transaction.Synchronization` Javadoc for details.

The `begin`, `commit`, and `rollback` methods demarcate transaction boundaries. The methods should be self-explanatory: `begin` starts a transaction, `commit` attempts to commit the transaction's changes to the data store, and `rollback` aborts the transaction, in which case the data store is "rolled back" to its previous state. JDO implementations will automatically roll back transactions if any fatal exception is thrown during the commit process. Otherwise, it is up to you to roll back the transaction to free its resources.

Finally, the `isActive` method returns true if the transaction is in progress (`begin` has been called more recently than `commit` or `rollback`), and false otherwise.

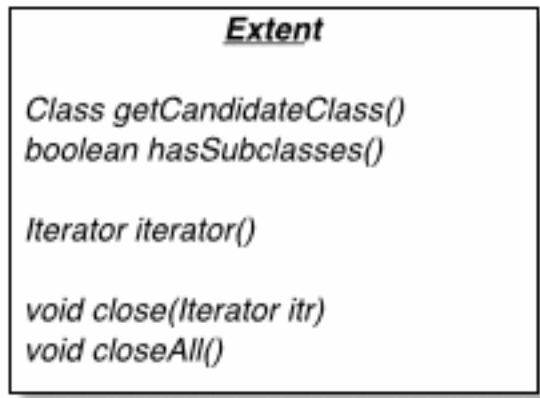
Example 9.1. Grouping Operations with Transactions

```

public void transferFunds (User from, User to, double amnt)
{
    PersistenceManager pm = JDOHelper.getPersistenceManager (from);
    Transaction trans = pm.currentTransaction ();
    try
    {
        trans.begin ();
        from.decrementAccount (amnt);
        to.incrementAccount (amnt);
        trans.commit ();
    }
    catch (JDOFatalException jfe)
    {
        throw jfe;
    }
    catch (RuntimeException re)    // includes normal JDO exceptions
    {
        trans.rollback ();        // or could attempt to fix error and retry
        throw re;
    }
}

```

Chapter 10. Extent



An *Extent* is a logical view of all persistent instances of a given persistence-capable class, possibly including subclasses. *Extents* are obtained from *PersistenceManagers*, and are usually used to specify the candidate objects to a *Query*.

The *Extent* interface is straightforward. The *getCandidateClass* method returns the persistence-capable class of the *Extent*'s instances. The *hasSubclasses* method indicates whether instances of subclasses are also included.

You can obtain an iterator over every object in an *Extent* using the *iterator* method. The iterators used by some implementations might consume data store resources; therefore, you should always close an *Extent*'s iterators as soon as you are done with them. You can close an individual iterator by passing it to the *close* method, or you can close all open iterators at once with *closeAll*.

Chapter 11. Query

```

Query

boolean ignoreCache

void setClass(Class pcClass)
void setCandidates(...)
void setFilter(String filter)

void declareImports(String imports)
void declareParameters(String parameters)
void declareVariables (String variables)

Object execute(...)
Object executeWithMap(Map params)
Object executeWithArray(Object[] params)

void close(Object result)
void closeAll()

```

You can obtain `Query` instances from a `PersistenceManager`. They are used to filter a set of candidate objects based on certain criteria. This filtering might take place in the data store, or might be executed in memory. JDO does not mandate any one query mechanism, and most implementations probably use a mixture of datastore and in-memory execution depending on the circumstances.

We will now explore the elements of the `Query` interface. You might find yourself scratching your head over some aspects of the discussion below; this is to be expected. All will be made clear when we review several examples of JDO queries later in this chapter.

11.1. Required Query Elements

Every `Query` has three required elements:

- The candidate class. Only instances of this class and its subclasses will be considered when evaluating the query filter. The candidate class is set through the `setClass` method.
- The set of candidate objects. Candidates can be specified as either a `Collection` of objects or as an `Extent`. There are `setCandidate` methods to handle both cases. If an `Extent` is given, then you do not need to separately specify a candidate class for the query; the query will inherit the `Extent`'s candidate class.
- The filter string. This is a `String` written in the JDO Query Language (JDOQL) and set via the `setFilter` method. If you do not specify a filter, all objects in the candidate set that are assignable to the candidate class will match the query.

11.2. Optional Query Elements

The following are optional elements of JDO queries:

- **Imports.** Imports are specified with the `declareImports` method, and are given as a single, semicolon-delimited string following the standard Java `import` syntax. They are used so that you don't have to type out full class names when declaring parameters and variables, as described below.
- **Parameter declarations.** Parameters act as placeholders in the filter string. They allow you to write a single query, then execute it multiple times, supplying new values each time. Parameters are specified via the `declareParameters` method. The syntax of the method's `String` argument follows the syntax for declaring the parameter signature of a Java method.
- **Variable declarations.** Variables are typically used to test whether some item in a collection matches certain criteria. They are specified with the `declareVariables` method, using the standard Java syntax for variable declarations.
- **Ordering.** Sometimes you would like the results of a query to be returned in a specific order. For example, you might want a list of all cars for sale in ascending order by price. The `setOrdering` method enables you to add ordering criteria to your queries. The `String` argument to the method is a comma-separated list of ordering declarations, where each declaration consists of a field name followed by the keywords "ascending" or "descending". The results will be ordered primarily by the first (left-most) ordering declaration. Wherever two results compare equal with this expression, the next ordering expression will be used to order them, and so on.

11.3. JDOQL

JDOQL is a data store-neutral query language based on Java boolean expressions. The syntax of JDOQL is the same as standard Java syntax, with the following exceptions:

- Equality and ordering comparisons between primitives and instances of wrapper classes (`Boolean`, `Byte`, `Integer`, etc) are valid.
- Equality and ordering between `Dates` are valid.
- The assignment operators (`=`, `+=`, `*=`, etc) and `++` and `--` operators are not supported.
- Methods are not supported, with the following exceptions:
 - `Collection.contains`
 - `Collection.isEmpty`
 - `String.startsWith`
 - `String.endsWith`
- Traversing a null-valued field, which would normally throw a `NullPointerException`, instead causes the subexpression to evaluate to `false` for the current candidate.
- The following literal types are supported: character literals, integer literals, floating point literals, boolean literals, string literals, and the `null` literal.

Note

Kodo JDO also supports the `Map.containsKey` and `Map.containsValue` methods.

We will now present several examples illustrating the features of JDOQL and the Query interface. The examples use the following persistence-capable classes:

```
package org.mag;

public class Magazine
{
    private String title;
    private double price;
    private int copiesSold;
    private Company publisher;
    private Article coverArticle;
    private Set articles;

    ...
}

public class Article
{
    private String title;
    private Collection subTitles;

    ...
}

package org.mag.pub;

public class Company
{
    private String name;
    private double revenue;

    ...
}
```

Example 11.1. Basic Query

Find all magazines whose price is greater than 3 dollars:

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "price > 10.0";
Query query = pm.newQuery (mags, filter);
Collection results = (Collection) query.execute ();
```

Example 11.2. Result Ordering and Method Calls

Find all magazines whose title starts with "The ", in ascending order by price:

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "title.startsWith (\"The \")";
Query query = pm.newQuery (mags, filter);
query.setOrdering ("price ascending");
Collection results = (Collection) query.execute ();
```

Example 11.3. Mathematical Operations and Relation Traversal

Find all magazines whose revenue is over 1% of the total revenue for the publisher, in descending order by publisher revenue and ascending order by publisher name:

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "price * copiesSold > publisher.revenue * .01";
Query query = pm.newQuery (mags, filter);
query.setOrdering ("publisher.revenue descending, publisher.name ascending");
Collection results = (Collection) query.execute ();
```

Example 11.4. Precedence and Logical Operators

Find all magazines published by Random House or Addison Wesley whose price is less than or equal to 3 dollars:

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "price <= 10.0 "
    + "&& (publisher.name == \"Random House\" "
    + "|| publisher.name == \"Addison Wesley\")";
Query query = pm.newQuery (mags, filter);
Collection results = (Collection) query.execute ();
```

Example 11.5. Imports and Parameters

Find all magazines published by a given company whose price is greater than a given number:

```
Company myCompany = ...;
Double myPrice = ...;

Extent mags = pm.getExtent (Magazine.class, false);
String filter = "publisher == pub && price > amnt";
Query query = pm.newQuery (mags, filter);
query.declareImports ("import org.mag.pub.*;");
query.declareParameters ("Company pub, Double amnt");
Collection results = (Collection) query.execute (myCompany, myPrice);
```

Example 11.6. Collections

Find all magazines whose cover article has a subtitle of "The Real Story" or whose cover article has no subtitles:

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "coverArticle.subTitles.contains (\"The Real Story\") "
    + "|| coverArticle.subTitles.isEmpty ()";
Query query = pm.newQuery (mags, filter);
Collection results = (Collection) query.execute ();
```

Example 11.7. Variables

Find all magazines that have an article titled "Fourier Transforms":

```
Extent mags = pm.getExtent (Magazine.class, false);
String filter = "articles.contains (art)
    && art.title == \"Fourier Transforms\"";
Query query = pm.newQuery (mags, filter);
query.declareVariables ("Article art;");
Collection results = (Collection) query.execute ();
```

11.4. Executing Queries

As evident from the examples above, queries are executed with one of the many `execute` methods defined in the `Query` interface. If your query declares between 0 and 3 parameters, use the `execute` version that takes the corresponding number of `Object` arguments; each argument should be set to a parameter value. For queries with more than 3 parameters, you can use the more generic `executeWithArray` and `executeWithMap` methods. See the `Query` interface Javadoc for details.

All `execute` methods declare a return type of `Object`, but they really return `Collections`. The JDO specification only uses `Object` rather than `Collection` to enable vendors to use proprietary return types in certain cases, and to allow for future expansion of the interface.

Query results may hold on to data store resources; therefore, all results should be closed when they are no longer needed. You can close an individual query result with the `close` method, or all open results at once with the `closeAll`.

11.5. Query Compilation

Query objects can be compiled via the `compile` method. Compiling a query is a hint to the implementation to optimize an execution plan for the query. During compilation, all query elements are validated; any inconsistencies are reported via a `JDOUserException`.

Query instances can also be serialized. After deserialization, a query cannot be executed, but it retains its candidate class, filter string, imports, parameters, variables, and ordering. A deserialized query can therefore act as a template to create other query instances with the same configuration. This is accomplished through the `PersistenceManager`'s `newQuery(Object template)` method.

Part III. Kodo JDO Reference Guide

Chapter 1. Introduction

Kodo JDO is a JDBC-based implementation of the JDO 1.0 standard for object persistence. This document is a reference for the configuration and use of Kodo JDO.

1.1. Intended Audience

This document is intended for Kodo JDO developers. It assumes strong knowledge of Java, familiarity with the eXtensible Markup Language (XML), and an understanding of JDO. If you are not familiar with JDO, please read SolarMetric's JDO Overview before proceeding.

Certain sections of this guide cover advanced topics such as custom object-relational mapping, enterprise integration, and using Kodo with third-party tools. These sections assume prior experience with the relevant subject.

Chapter 2. Configuration Framework

Kodo JDO implements a unified configuration framework across its runtime environment and all of its development tools. The framework is based on SolarMetric's `com.solarmetric.kodo.conf.Configuration` interface. Concrete implementations of this interface and its subclasses are freely interchangeable with standard Java `Properties` objects, so most of your configuration will be through properties objects or properties files. For extreme customization, however, Kodo JDO's `PersistenceManagerFactory` implementation and its development tools allow you to manipulate the `Configuration` instance directly. See their Javadoc for details.

In JDO 1.0, `PersistenceManagerFactory`s are usually obtained through the `JDOHelper.getPersistenceManagerFactory` method, which takes a single `Properties` argument. The supplied properties are used to configure the JDO implementation. Similarly, all Kodo JDO development tools accept a `-properties` command-line flag specifying the location of a properties file to read from. This location can be given as either a path to a file, or as a resource name of a file somewhere in the `CLASSPATH`. Kodo JDO's tools use the same property keys as the runtime environment, so you can use the same properties files for both development and runtime.

The development tools and runtime environment also share a comprehensive system of property defaults and overrides defined by the `Configuration` interface:

- All properties are defaulted to the values specified in an optional `kodo.properties` resource that can be placed in any top-level directory of the `CLASSPATH`.
- You can customize the location of the above resource by specifying the correct resource name in the `com.solarmetric.kodo.properties` System property.
- For users of the deprecated `system.prefs` configuration mechanism present in earlier versions of Kodo JDO, the information in your `system.prefs` will also be loaded as default values.
- You can override any default value defined in the `kodo.properties` resource by setting the System property of the same name to the desired default value.
- All Kodo JDO command-line tools accept flags to set the value of any property. The flag name is always the last token of the corresponding property name, capitalized as a Java identifier. For example, to set the JDO `javax.jdo.option.ConnectionUserName` property, you could pass the `-connectionUserName <value>` flag to any tool.

Kodo JDO also enables the creation and configuration of system plugins via properties. Plugin-related properties typically come in pairs: `com.solarmetric.kodo.<property>Class` and `com.solarmetric.kodo.<property>Properties`. The `<property>Class` key is used to specify the plugin's class, and the `<property>Properties` key is used to configure the plugin instance once it is instantiated. This configuration is automatic. Kodo JDO matches the keys supplied in the `<property><Properties` string to the setter methods of the plugin using Java bean naming conventions. The string must be of the form "`<key1>=<value1> <key2>=<value2> ...`". Consider the following example:

Suppose that you have created a new class, `com.xyz.MyDataCache`, that you wish to use in Kodo JDO's pluggable caching mechanism. `MyDataCache` has two configuration methods, `setMaxSize (int)` and `setRemoteHost (String, int)`. You could plug your cache into Kodo JDO by specifying the following properties:

```
com.solarmetric.kodo.DataCacheClass: com.xyz.MyDataCache
com.solarmetric.kodo.DataCacheProperties: maxSize=100 remoteHost=CacheServer,8080
```

2.1. JDO Standard Properties

JDO recognizes many standard runtime properties, all of which Kodo JDO supports (these properties are also covered in the JDO Overview).

2.1.1. `javax.jdo.PersistenceManagerFactoryClass`

Property name: `javax.jdo.PersistenceManagerFactoryClass`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getPersistenceManagerFactoryClass`

Resource adaptor config-property: `PersistenceManagerFactoryClass`

Default: `none`

Description: The name of the concrete implementation of `javax.jdo.PersistenceManagerFactory` that `javax.jdo.JDOHelper.getPersistenceManagerFactory()` should create. For Kodo JDO, this should be `com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory` or `com.solarmetric.kodo.impl.jdbc.ee.EEPersistenceManagerFactory`, or a custom extension of one of these types.

2.1.2. `javax.jdo.option.Optimistic`

Property name: `javax.jdo.option.Optimistic`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getOptimistic`

Resource adaptor config-property: `Optimistic`

Default: `true`

Description: Selects between optimistic and pessimistic (data store) transactional modes.

2.1.3. `javax.jdo.option.Multithreaded`

Property name: `javax.jdo.option.Multithreaded`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getMultithreaded`

Resource adaptor config-property: `Multithreaded`

Default: `false`

Description: If `true`, then the application plans to have multiple threads simultaneously accessing a single `PersistenceManager`, so measures must be taken to ensure that the implementation is thread-safe. Otherwise, the implementation need not address thread safety.

2.1.4. `javax.jdo.option.IgnoreCache`

Property name: `javax.jdo.option.IgnoreCache`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getIgnoreCache`

Resource adaptor config-property: `IgnoreCache`

Default: `false`

Description: If `false`, then the JDO implementation must consider modifications, deletions, and additions in the `PersistenceManager` transaction cache when executing a query inside a transaction. Else, the implementation is free to ignore the cache and execute the query directly against the data store.

2.1.5. `javax.jdo.option.RetainValues`

Property name: `javax.jdo.option.RetainValues`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getRetainValues`

Resource adaptor config-property: `RetainValues`

Default: `true`

Description: If `true`, then fields in a persistence-capable object that have been read during a transaction must be preserved in memory after the transaction commits. Otherwise, persistence-capable objects must transition to the hollow state upon commit, meaning that subsequent reads will result in a database round-trip.

2.1.6. `javax.jdo.option.RestoreValues`

Property name: `javax.jdo.option.RestoreValues`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getRestoreValues`

Resource adaptor config-property: `RestoreValues`

Default: `true`

Description: If `true`, then fields in a persistence-capable object that have been changed during a transaction will be rolled back to their original values upon a `rollback`. Otherwise, the values will not be changed upon `rollback`.

2.1.7. `javax.jdo.option.NontransactionalRead`

Property name: `javax.jdo.option.NontransactionalRead`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getNontransactionalRead`

Resource adaptor config-property: `NontransactionalRead`

Default: `true`

Description: If `true`, then it is possible to read persistent data outside the context of a transaction. Otherwise, a transaction must be in progress in order read data.

2.1.8. `javax.jdo.option.NontransactionalWrite`

Property name: `javax.jdo.option.NontransactionalWrite`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getNontransactionalWrite`

Resource adaptor config-property: `NontransactionalWrite`

Default: `false`

Description: If `true`, then it is possible to write to fields of a persistent-nontransactional object when a transaction is not in progress. If `false`, such a write will result in a `JDOUserException`.

2.1.9. `javax.jdo.option.ConnectionURL`

Property name: `javax.jdo.option.ConnectionURL`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionURL`

Resource adaptor config-property: `ConnectionURL`

Default: none

Description: The URL for the data source.

2.1.10. `javax.jdo.option.ConnectionUserName`

Property name: `javax.jdo.option.ConnectionUserName`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionUserName`

Resource adaptor config-property: `ConnectionUserName`

Default: none

Description: The username for the connection listed in `ConnectionURL`.

2.1.11. `javax.jdo.option.ConnectionPassword`

Property name: `javax.jdo.option.ConnectionPassword`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionPassword`

Resource adaptor config-property: `ConnectionPassword`

Default: none

Description: The password for the user specified in `ConnectionUserName`

2.1.12. `javax.jdo.option.ConnectionDriverName`

Property name: `javax.jdo.option.ConnectionDriverName`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionDriverName`

Resource adaptor config-property: `ConnectionDriverName`

Default: none

Description: The class name of either the JDBC `java.sql.Driver`, or an instance of a `javax.sql.DataSource` to use to connect to the data source.

2.1.13. `javax.jdo.option.ConnectionFactoryName`

Property name: `javax.jdo.option.ConnectionFactoryName`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionFactoryName`

Resource adaptor config-property: `ConnectionFactoryName`

Default: none

Description: The JNDI name of the connection factory to use for obtaining connections.

2.1.14. `javax.jdo.option.ConnectionFactory2Name`

Property name: `javax.jdo.option.ConnectionFactory2Name`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionFactory2Name`

Resource adaptor config-property: `ConnectionFactory2Name`

Default: none

Description: The JNDI name of the connection factory to use for finding read-only connections.

2.1.15. `javax.jdo.option.MinPool`

Property name: `javax.jdo.option.MinPool`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getMinPool`

Resource adaptor config-property: `MinPool`

Default: 0

Description: The minimum number of connections to keep in the pool. This option has been removed from the specification, but we still use the `javax.jdo.option` for backwards compatibility.

2.1.16. `javax.jdo.option.MaxPool`

Property name: `javax.jdo.option.MaxPool`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getMaxPool`

Resource adaptor config-property: `MaxPool`

Default: 10

Description: The maximum number of connections to pool. If all of these are in use, then `PersistenceManager` instances must wait for a connection to become available. This option has been removed from the specification, but we still use the `javax.jdo.option` for backwards compatibility.

2.1.17. `javax.jdo.option.MsWait`

Property name: `javax.jdo.option.MsWait`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getMsWait`

Resource adaptor config-property: MsWait

Default: 0

Description: The number of milliseconds to wait for a pooled connection before throwing an exception if the pool is empty. This option has been removed from the specification, but we still use the `javax.jdo.option` for backwards compatibility.

2.2. Kodo JDO Properties

Kodo JDO defines many properties of its own. Most of these properties are provided for advanced users who wish to customize Kodo JDO's behavior; the majority of developers can omit them. A complete listing of Kodo JDO-specific properties is given below.

2.2.1. `com.solarmetric.kodo.CacheReferenceSize`

Property name: `com.solarmetric.kodo.CacheReferenceSize`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getCacheReferenceSize`

Resource adaptor config-property: `CacheReferenceSize`

Default: 1000

Description: The number of hard references to cached objects that the `PersistenceManager`'s cache will retain (in addition to the soft reference cache that it maintains). Setting this to a higher value will result in more objects being retained in the cache, at the cost of utilizing more memory resources. Setting this to -1 will cause all loaded objects to have hard references.

2.2.2. `com.solarmetric.kodo.DataCacheClass`

Property name: `com.solarmetric.kodo.DataCacheClass`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getDataCacheClass`

Resource adaptor config-property: `DataCacheClass`

Default: none

Description: The name of the class to use for caching of data loaded from the data store. Must implement `com.solarmetric.kodo.runtime.datacache.DataCache`.

2.2.3. `com.solarmetric.kodo.ConnectionProperties`

Property name: `com.solarmetric.kodo.ConnectionProperties`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionProperties`

Resource adaptor config-property: `ConnectionProperties`

Default: none

Description: A space-separated list of properties to be passed to the JDBC Driver when obtaining a Connection.

Properties are of the form "*key=value*".

Example 2.1. Specifying Connection Properties

```
com.solarmetric.kodo.ConnectionProperties: serverName=myserver port=1266
```

If a `javax.sql.DataSource` class is defined in the `javax.jdo.option.ConnectionDriverName` property, then this property will be used to set bean-like properties in the `DataSource` instance upon creation. These properties vary depending on the `DataSource` in use: see the documentation for your `DataSource` for details on the properties to use.

Example 2.2. Specifying DataSource Properties

```
javax.jdo.option.ConnectionDriverName=oracle.jdbc.pool.OracleDataSource
com.solarmetric.kodo.ConnectionProperties=PortNumber=1521 \
    ServerName=saturn \
    DatabaseName=solarsid \
    DriverType=thin
```

2.2.4. com.solarmetric.kodo.DataCacheProperties

Property name: `com.solarmetric.kodo.DataCacheProperties`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getDataCacheProperties`

Resource adaptor config-property: `DataCacheProperties`

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.DataCacheClass` upon initialization. See the caching documentation for details on possible values.

2.2.5. com.solarmetric.kodo.DefaultFetchThreshold

Property name: `com.solarmetric.kodo.DefaultFetchThreshold`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getDefaultFetchThreshold`

Resource adaptor config-property: `DefaultFetchThreshold`

Default: 30

Description: The threshold below which result lists will be completely instantiated upon their creation. A value of -1 will always force all results to be completely instantiated, thus disabling lazy result loading.

2.2.6. com.solarmetric.kodo.DefaultFetchBatchSize

Property name: `com.solarmetric.kodo.DefaultFetchBatchSize`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getDefaultFetchBatchSize`

Resource adaptor config-property: DefaultFetchBatchSize

Default: 10

Description: The number of rows that will be pre-fetched when an element in a Query result is accessed.

2.2.7. com.solarmetric.kodo.EnableQueryExtensions

Property name: com.solarmetric.kodo.EnableQueryExtensions

Configuration API: com.solarmetric.kodo.conf.Configuration.getEnableQueryExtensions

Resource adaptor config-property: EnableQueryExtensions

Default: false

Description: If true, then Kodo JDO will allow the use of query filter extensions. See the query extensions documentation for more information.

2.2.8. com.solarmetric.kodo.LicenseKey

Property name: com.solarmetric.kodo.LicenseKey

Configuration API: com.solarmetric.kodo.conf.Configuration.getLicenseKey

Resource adaptor config-property: LicenseKey

Default: none

Description: The license key provided to you by SolarMetric. Keys are available at www.solarmetric.com

2.2.9. com.solarmetric.kodo.PersistenceManagerClass

Property name: com.solarmetric.kodo.PersistenceManagerClass

Configuration API: com.solarmetric.kodo.conf.Configuration.getPersistenceManagerClass

Resource adaptor config-property: PersistenceManagerClass

Default: com.solarmetric.kodo.runtime.PersistenceManagerImpl

Description: The name of the class that the PersistenceManagerFactory should create when creating a new PersistenceManagerImpl. Must extend com.solarmetric.kodo.runtime.PersistenceManagerImpl.

2.2.10. com.solarmetric.kodo.PersistenceManagerProperties

Property name: com.solarmetric.kodo.PersistenceManagerProperties

Configuration API: com.solarmetric.kodo.conf.Configuration.getPersistenceManagerProperties

Resource adaptor config-property: PersistenceManagerProperties

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.PersistenceManagerClass` upon initialization.

2.2.11. `com.solarmetric.kodo.ProxyManagerClass`

Property name: `com.solarmetric.kodo.ProxyManagerClass`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getProxyManagerClass`

Resource adaptor config-property: `ProxyManagerClass`

Default: `com.solarmetric.kodo.util.SimpleProxyManager`

Description: The name of the class to use to proxy second class objects in managed instances. Must implement `com.solarmetric.kodo.util.ProxyManager`.

2.2.12. `com.solarmetric.kodo.ProxyManagerProperties`

Property name: `com.solarmetric.kodo.ProxyManagerProperties`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getProxyManagerProperties`

Resource adaptor config-property: `ProxyManagerProperties`

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.ProxyManagerClass` upon initialization.

2.2.13. `com.solarmetric.kodo.QueryFilterListeners`

Property name: `com.solarmetric.kodo.QueryFilterListeners`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getQueryFilterListeners`

Resource adaptor config-property: `QueryFilterListeners`

Default: none

Description: A list of query filter listeners to add to the default list of extensions. Ignored if `com.solarmetric.kodo.EnableQueryExtensions` is false.

2.2.14. `com.solarmetric.kodo.UseSoftTransactionCache`

Property name: `com.solarmetric.kodo.UseSoftTransactionCache`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getUseSoftTransactionCache`

Resource adaptor config-property: `UseSoftTransactionCache`

Default: false

Description: Sets whether or not Kodo should maintain soft references to transactional items that have not been dirtied. This can be useful in esoteric situations involving iterating through a massive list of objects within a

transaction but only dirtying a couple towards the end of the list. However, there is a noticeable performance impact involved with turning this option on, in particular when loading massive lists of objects into a transaction.

2.2.15. com.solarmetric.kodo.PersistentTypes

Property name: `com.solarmetric.kodo.PersistentTypes`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getPersistentTypes`

Resource adaptor config-property: `PersistentTypes`

Default: `none`

Description: A comma-separated list of classes that are to be instantiated whenever a new `PersistenceManager` is created. This property is optional, but it can be used to get around known deficiencies in the JDO specification whereby locating a persistent instance based in an application id class is not possible until the target persistent class has been loaded by the JVM. This property is also used to optimize datastore subclass identification: normally, subclass location is done by issuing a "SELECT DISTINCT [subclass identifier column]" statement. This property makes that identification unnecessary, which can lead to some performance benefits at initialization time when for very large tables. If this property is set, then all the persistent types in the system must be enumerated; failure to do so will lead to a warning, and may result in a failure to correctly locate subclasses for a persistent inheritance model.

2.2.16. com.solarmetric.kodo.impl.jdbc.ConnectionTestTimeout

Property name: `com.solarmetric.kodo.impl.jdbc.ConnectionTestTimeout`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getConnectionTestTimeout`

Resource adaptor config-property: `ConnectionTestTimeout`

Default: `10`

Description: The number of seconds to wait between testing connections retrieved from the connection pool. Only valid when using the built-in Kodo connection pooling.

2.2.17. com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass

Property name: `com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getDefaultClassMappingClass`

Resource adaptor config-property: `DefaultClassMappingClass`

Default: `com.solarmetric.kodo.impl.jdbc.ormapping.ClassMapping`.

Description: The name of the default class to use for mapping persistent classes to the database. Must extend `com.solarmetric.kodo.impl.jdbc.ormapping.ClassMapping`.

2.2.18. com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass

Property name: `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getDefaultSubclassProviderClass`

Resource adaptor config-property: DefaultSubclassProviderClass

Default: `com.solarmetric.kodo.impl.jdbc.ormapping.SubclassProviderImpl`. This implementation stores the full classname of each type stored into the database in the indicator column defined in the JDO metadata file.

Description: The name of the default class to use for managing subclass indicator columns. Must implement the `com.solarmetric.kodo.impl.jdbc.ormapping.SubclassProvider` interface. See custom class indicator documentation for more information about subclass providers.

2.2.19.

`com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderProperties`

Property name: `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderProperties`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getDefaultSubclassProviderProperties`

Resource adaptor config-property: DefaultSubclassProviderProperties

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass` upon initialization.

2.2.20. `com.solarmetric.kodo.impl.jdbc.DictionaryClass`

Property name: `com.solarmetric.kodo.impl.jdbc.DictionaryClass`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getDictionaryClass`

Resource adaptor config-property: DictionaryClass

Default: based on `javax.jdo.option.ConnectionURL`

Description: The `DBDictionary` to use for this configuration. This is auto-detected based on the setting of `javax.jdo.option.ConnectionURL`, so you need only set this to override the default with your own custom `DBDictionary` or if you are using an unrecognized driver.

Unfortunately, minor differences in the way databases map java types to native SQL types and variances in the SQL syntax for manipulating database schema make it impossible to write persistence code that will work across all databases. To overcome this problem, SolarMetric has defined the `com.solarmetric.kodo.impl.jdbc.schema.DBDictionary` interface, which declares the API necessary to abstract away the idiosyncrasies of an individual database vendor. Each supported database has its own dictionary:

- `com.solarmetric.kodo.impl.jdbc.schema.dict.CloudscapeDictionary`
- `com.solarmetric.kodo.impl.jdbc.schema.dict.DB2Dictionary`
- `com.solarmetric.kodo.impl.jdbc.schema.dict.HSQLDictionary`
- `com.solarmetric.kodo.impl.jdbc.schema.dict.InstantDBDictionary`
- `com.solarmetric.kodo.impl.jdbc.schema.dict.MySQLDictionary`

- `com.solarmetric.kodo.impl.jdbc.schema.dict.OracleDictionary`
 - `com.solarmetric.kodo.impl.jdbc.schema.dict.PointbaseDictionary`
 - `com.solarmetric.kodo.impl.jdbc.schema.dict.PostgresDictionary`
 - `com.solarmetric.kodo.impl.jdbc.schema.dict.SQLServerDictionary`
 - `com.solarmetric.kodo.impl.jdbc.schema.dict.SybaseDictionary`
- For databases without direct support, it is possible to implement a custom `DBDictionary` and plug it in using this property. To facilitate this process, SolarMetric provides the `com.solarmetric.kodo.impl.jdbc.schema.dict.GenericDictionary`, which implements the `DBDictionary` interface using standard SQL. Subclasses need override only those methods that represent operations for which a specific database differs from the standard. See the included Javadoc documentation for further information.

2.2.21. `com.solarmetric.kodo.impl.jdbc.DictionaryProperties`

Property name: `com.solarmetric.kodo.impl.jdbc.DictionaryProperties`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getDictionaryProperties`

Resource adaptor config-property: `DictionaryProperties`

Default: none

Description: A space-separated list of name-value properties settings to pass to the dictionary defined by `com.solarmetric.kodo.impl.jdbc.DictionaryClass`.

Many of the `DBDictionary` options are automatically configured by concrete subclasses of `GenericDictionary`. The defaults can, however, be overridden by using this property.

The `GenericDictionary` understands the following properties:

- `QuoteNumbers`

Default: `false`

Description: If `true`, then numbers will be quoted before being stored. This improves precision for some data stores.

- `NameTruncationVersion`

Default: `0`

Description: If `0`, then autogenerated table/column/index names that are longer than the maximum allowable lengths will be truncated using a readable truncation algorithm that can potentially cause collisions. If `1`, then a more aggressive but less readable truncation algorithm will be used. This is useful for databases with small maximum lengths, such as DB2. For information on manually overriding Kodo JDO's schema name generation to avoid collisions, see the section on JDO metadata extensions.

- `SchemaName`

Default: `null`

Description: The name of the schema to connect to when invoking `java.sql.DatabaseMetaData` methods that accept a schema name. This can be important when accessing a data store that has multiple

schemas with JDO metadata tables in them.

- `IndexNameGenerator`

Default: `com.solarmetric.kodo.impl.jdbc.schema.DefaultNameGenerator`

Description: The concrete implementation of the `com.solarmetric.kodo.impl.jdbc.schema.NameGenerator` interface, which controls how index names will be generated.

- `ColumnNameGenerator`

Default: `com.solarmetric.kodo.impl.jdbc.schema.DefaultNameGenerator`

Description: The concrete implementation of the `com.solarmetric.kodo.impl.jdbc.schema.NameGenerator` interface, which controls how column names will be generated.

- `TableNameGenerator`

Default: `com.solarmetric.kodo.impl.jdbc.schema.DefaultNameGenerator`

Description: The concrete implementation of the `com.solarmetric.kodo.impl.jdbc.schema.NameGenerator` interface, which controls how table names will be generated.

- `SimulateLocking`

Default: `false`

Description: Some databases do not support pessimistic locking, which will result in a `JDOException` to be thrown when a datastore transaction is attempted. Setting this parameter to `true` will bypass this check, and allow a datastore transaction to occur even though the underlying database does not support them.

- `ValidateConnections`

Default: `true`

Description: Many JDBC drivers do not actually validate the `Connection` when `Connection.isClosed()` is issued. When set to `true`, the dictionary will perform additional validation of a JDBC `Connection` when retrieving a `Connection` from the `DataSource`. This will typically take the form of the issuance of a small SQL statement (such as "SELECT SYSDATE FROM DUAL"). This helps to ensure that `Connection` instances retrieved from a connection pool are in a valid state.

- `ValidateConnectionSQL`

Default: `null`

Description: A SQL statement to issue that a `Connection` instance is in a valid state.

- `StoreLargeNumbersAsStrings`

Default: `false`

Description: Many databases have limitation on the number of digits can can be stored in a numeric field (for example, Oracle can only store 38 digits). For applications that may be operating on very large `BigInteger` and `BigDecimal` values, it may be necessary to store these objects as `String` fields rather than the database's numeric type. Note that this may prevent meaningful numeric queries from being executed against the database.

The `MySQLDictionary` understands the following properties:

- `TableType`

Default: `null`

Description: The table type to use when creating new tables. For example, to create transaction-capable BDB tables, set this to `BDB`.

- `SupportsSelectForUpdate`

Default: `true`

Description: If `true`, then assume that this MySQL install is capable of locking data on select, using `SELECT ... FOR UPDATE` syntax. Otherwise, assume that this MySQL install cannot lock data. Currently, Kodo JDO silently ignores requests to obtain pessimistic locks in this situation. In the future, Kodo JDO will throw an exception if configured with data store exceptions while this is `false`.

2.2.22. `com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping`

Property name: `com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getFlatInheritanceMapping`

Resource adaptor config-property: `FlatInheritanceMapping`

Default: `true`

Description: If `true`, then all fields of all classes in a given inheritance hierarchy will by default map into the least-derived type's default primary table. If `false` then a new default primary table will be created for each class in the inheritance hierarchy, and each type's declared fields will map to that table by default.

2.2.23. `com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass`

Property name: `com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getSequenceFactoryClass`

Resource adaptor config-property: `SequenceFactoryClass`

Default: `com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory`. This implementation obtains sequence numbers from a special database table used solely for this purpose. The table is created automatically the first time sequence numbers are requested if it does not already exist. The sequence table can also be manipulated through the included `sequencetable` shell/bat script, which accepts the same options as the `schematool` and uses the following actions:

- `add`: Creates the sequence table if it does not already exist.
- `drop`: Drops the sequence table.
- `increment`: Increments the value of the sequence table by the increment defined for the `SequenceFactory`, and prints the new value. If the table does not exist, it will be created.

Description: The name of the class to use for generating sequence numbers when using data store identity. Must implement the `com.solarmetric.kodo.impl.jdbc.SequenceFactory` interface. Simple examples are

included in the source directory of your Kodo Installation. In addition there is also `com.solarmetric.kodo.impl.jdbc.schema.ClassDBSequenceFactory`, which is a class-sensitive `SequenceFactory`. This takes the same options as `DBSequenceFactory`, as well as having a `main (String [])` to manipulate the factory.

2.2.24. `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties`

Property name: `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getSequenceFactoryProperties`

Resource adaptor config-property: `SequenceFactoryProperties`

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass` upon initialization.

`com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory` understands the following properties:

- Increment

Default: 50

Description: The number by which the sequence table should be incremented. The sequence table is created automatically by Kodo JDO and used to generate unique object ID values. To increase performance, Kodo JDO grabs sequence numbers in blocks, so that it only has to consult the sequence table once every *N* new persistent instances. The default value for this property is 50.

- TableName

Default: JDO_SEQUENCE

Description: The sequence table to look for the sequence values. The default value for this property is JDO_SEQUENCE. Note that tables names will go through the `DBDictionary` for their ultimate table name.

2.2.25. `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass`

Property name: `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getSQLExecutionManagerClass`

Resource adaptor config-property: `SQLExecutionManagerClass`

Default: `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerImpl`

Description: The name of a custom `SQLExecutionManager` to be used for all issuance of SQL to the data store. Must implement `com.solarmetric.kodo.impl.jdbc.SQLExecutionManager`.

2.2.26. `com.solarmetric.kodo.impl.jdbc.StatementCacheMaxSize`

Property name: `com.solarmetric.kodo.impl.jdbc.StatementCacheMaxSize`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getStatementCacheMaxSize`

Resource adaptor config-property: `StatementCacheMaxSize`

Default: 100

Description: The size of the `PreparedStatement` cache that is maintained in the `DataSource` implementation.

2.2.27. `com.solarmetric.kodo.impl.jdbc.StatementExecutionTimeout`

Property name: `com.solarmetric.kodo.impl.jdbc.StatementExecutionTimeout`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getStatementExecutionTimeout`

Resource adaptor config-property: `StatementExecutionTimeout`

Default: -1

Description: The time, in seconds, after which a JDBC query will be aborted if it has not yet returned any values. This value is simply passed to the JDBC driver's `Statement.setTimeout` method; Kodo does not perform any addition timeout actions. Note that many JDBC drivers either ignore this request, or improperly handle it, which may result in application deadlocks.

2.2.28. `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema`

Property name: `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getSynchronizeSchema`

Resource adaptor config-property: `SynchronizeSchema`

Default: `false`

Description: If `true`, the Kodo runtime will automatically attempt to refresh the database schema when persistent classes are referenced, allowing the developer to bypass the `schematool` step.

Warning

This property is only intended to be used for development. As automatic schema migration can result in data loss, this feature should never be enabled on a production system. Furthermore, this feature has serious adverse affects on Kodo's runtime performance. Ensure that it is disabled before doing any performance analysis.

2.2.29. `com.solarmetric.kodo.impl.jdbc.TransactionIsolation`

Property name: `com.solarmetric.kodo.impl.jdbc.TransactionIsolation`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getTransactionIsolation`

Resource adaptor config-property: `TransactionIsolation`

Default: `none`

Description: Typically, the default transaction isolation provided by a `java.sql.Connection` will be appropriate for an application. However, it is sometimes desirable to override the default transaction isolation. This property can be used to set the transaction isolation for every transactional connection that is made to the data store.

Valid values are:

- `READ_COMMITTED`: dirty reads are prevented; non-repeatable reads and phantom reads can occur
- `READ_UNCOMMITTED`: dirty reads, 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

2.2.30. `com.solarmetric.kodo.impl.jdbc.UsePreparedStatements`

Property name: `com.solarmetric.kodo.impl.jdbc.UsePreparedStatements`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getUsePreparedStatements`

Resource adaptor config-property: `UsePreparedStatements`

Default: `true`

Description: Use a `PreparedStatement` for all SQL access.

2.2.31. `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements`

Property name: `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getUseBatchedStatements`

Resource adaptor config-property: `UseBatchedStatements`

Default: `true` (provided the underlying JDBC driver's method `supportsBatchUpdates()` returns `true`)

Description: Whenever possible, batch together similar non-selecting SQL statements (INSERT/UPDATE/DELETE) for performance.

2.2.32. `com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure`

Property name: `com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCCConfiguration.getWarnOnPersistentTypeFailure`

Resource adaptor config-property: `WarnOnPersistentTypeFailure`

Default: `false`

Description: If `true`, then Kodo JDO will print a warning if an error occurs while loading one of the listed persistent types. Otherwise, Kodo JDO will fail when such an error occurs. This can be useful when developing in a multi-user environment, when the schema may be slightly out-of-sync. However, this option should not be used at deploy time, since at deploy time, any problem loading a listed persistent type is probably a big deal.

2.2.33. `com.solarmetric.kodo.ee.ManagedRuntimeClass`

Property name: `com.solarmetric.kodo.ee.ManagedRuntimeClass`

Configuration API: `com.solarmetric.kodo.ee.EEConfiguration.getManagedRuntimeClass`

Resource adaptor config-property: `ManagedRuntimeClass`

Default: `com.solarmetric.kodo.ee.JNDIManagedRuntime`

Description: The name of the class to use for obtaining a reference to the transaction manager in an enterprise environment. Must implement the `com.solarmetric.kodo.ee.ManagedRuntime` interface.

2.2.34. `com.solarmetric.kodo.ee.ManagedRuntimeProperties`

Property name: `com.solarmetric.kodo.ee.ManagedRuntimeProperties`

Configuration API: `com.solarmetric.kodo.ee.EEConfiguration.getManagedRuntimeProperties`

Resource adaptor config-property: `ManagedRuntimeProperties`

Default: none

Description: A space-separated list of properties to pass to the class defined in `com.solarmetric.kodo.ManagedRuntimeClass` upon initialization.

`com.solarmetric.kodo.ee.JNDIManagedRuntime` understands the following property:

- `TransactionManagerName`

Default: `java:/TransactionManager`

Description: The location in JNDI of the `javax.transaction.TransactionManager` to use to synchronize with a global transaction.

2.3. Logging Framework

Kodo JDO uses the Apache Jakarta Commons Logging thin library for issuing log messages. The Commons Logging libraries act as a wrapper around a number of popular logging APIs, including the Jakarta Log4J project, and the native `java.util.logging` package in JDK 1.4. If neither of these libraries are available, then logging will fall back to using a very simple console logging. The remainder of this section presumes that Log4J will be used for logging. For details on customization of the Commons project, or on details on any of the underlying logging packages, please see the appropriate project page.

Logging is done over a number of logging channels, each of which has a logging level, which controls the verbosity of log messages that are sent to the channel. Following is an overview of the logging channels that Kodo will use, with a summary of the different levels to which log messages will be sent.

- `com.solarmetric.kodo.Metadata`: Information about the parsing of JDO metadata will be sent to the `trace` level of this channel. Warnings about potential problems with metadata will be sent to the `warn` channel.
- `com.solarmetric.kodo.Enhance`: Messages issued by the JDO enhancer will be sent to this logger, on a variety of channels.
- `com.solarmetric.kodo.Runtime`: General Kodo runtime messages will be sent to this channel.
- `com.solarmetric.kodo.Configuration`: Information about Kodo Configuration will be sent to this

channel.

- `com.solarmetric.kodo.impl.jdbc.JDBC`: JDBC connection information will be sent to this channel.
- `com.solarmetric.kodo.impl.jdbc.SQL`: This is the most common logging channel to use. Detailed information about the execution of SQL statements and connections will be sent to the `info` channel. It is useful to enable this channel if you are curious about the exact SQL that Kodo issues to the data store. By default, the SQL being executed is only listed once, when a `PreparedStatement` is first created. When the statement is executed, a hashcode will be printed out. This number can be used to cross-reference the actual SQL string being executed. If this channel is set to `TRACE`, then the actual SQL executed will be printed out every time the statement is reused.

Note

Verbose SQL information is sent to this channel only when using Kodo's own pooling `DataSource` implementation. When using a custom `DataSource`, consult the documentation for that `DataSource` for details on how to enable logging messages.

- `com.solarmetric.kodo.impl.jdbc.Schema`: Details about the operation of the `SchemaTool` will be sent to this logging channel.

Example 2.3. Example `log4j.properties` file for moderately verbose logging

```
log4j.rootCategory=WARN, console
log4j.category.com.solarmetric.kodo.impl.jdbc.SQL=WARN, console
log4j.category.com.solarmetric.kodo.impl.jdbc.JDBC=WARN, console
log4j.category.com.solarmetric.kodo.impl.jdbc.Schema=INFO, console
log4j.category.com.solarmetric.kodo.Performance=INFO, console
log4j.category.com.solarmetric.kodo.Metadata=WARN, console
log4j.category.com.solarmetric.kodo.Enhance=WARN, console
log4j.category.com.solarmetric.kodo.Query=WARN, console
log4j.category.com.solarmetric.kodo.Runtime=INFO, console

log4j.appender.console=org.apache.log4j.ConsoleAppender
```

Example 2.4. Example `log4j.properties` file for disabled logging

```
log4j.rootCategory=ERROR, console
log4j.category.com.solarmetric.kodo.impl.jdbc.SQL=ERROR, console
log4j.category.com.solarmetric.kodo.impl.jdbc.JDBC=ERROR, console
log4j.category.com.solarmetric.kodo.impl.jdbc.Schema=ERROR, console
log4j.category.com.solarmetric.kodo.Performance=ERROR, console
log4j.category.com.solarmetric.kodo.Metadata=ERROR, console
log4j.category.com.solarmetric.kodo.Enhance=ERROR, console
log4j.category.com.solarmetric.kodo.Query=ERROR, console
log4j.category.com.solarmetric.kodo.Runtime=ERROR, console

log4j.appender.console=org.apache.log4j.ConsoleAppender
```

Example 2.5. Example log4j.properties file for debugging logging

```
log4j.rootCategory=TRACE, console
log4j.category.com.solarmetric.kodo.impl.jdbc.SQL=TRACE, console
log4j.category.com.solarmetric.kodo.impl.jdbc.JDBC=TRACE, console
log4j.category.com.solarmetric.kodo.impl.jdbc.Schema=TRACE, console
log4j.category.com.solarmetric.kodo.Performance=TRACE, console
log4j.category.com.solarmetric.kodo.Metadata=TRACE, console
log4j.category.com.solarmetric.kodo.Enhance=TRACE, console
log4j.category.com.solarmetric.kodo.Query=TRACE, console
log4j.category.com.solarmetric.kodo.Runtime=TRACE, console

log4j.appender.console=org.apache.log4j.ConsoleAppender
```

Chapter 3. Creating Persistent Classes

Persistent class basics are covered in the JDO Overview. This chapter details the tools Kodo JDO provides to aid in persistent class creation, and tips on how to optimize your classes for the Kodo JDO runtime.

3.1. Application Identity Class Generation

Kodo JDO supports both datastore and application JDO identity types. If you choose to use application identity, you may want to take advantage of Kodo JDO's `appidtool`.

The `appidtool` generates Java code implementing the object identity class for any persistent type using application identity. The code satisfies all the requirements JDO places on object identity classes. You can use it as-is, or simply use it as a starting point, editing it to meet your needs.

Before you can run the `appidtool` for a persistent class, the class must be compiled and must have JDO metadata. The metadata should include the `objectid-class`, though whatever class it refers to obviously may not exist yet. The tool will use the attribute to name the identity class that it generates.

The `appidtool` can be invoked via the included `appidtool` script or via its java class, `com.solarmetric.kodo.enhance.ApplicationIdTool`. It accepts the standard set of command-line arguments defined by the configuration framework. It also accepts an `-ignoreErrors` flag. If this flag is set to `false`, an exception will be thrown if the tool is run on any class that does not use application identity, or is not the base class in the inheritance hierarchy (subclasses never define the application identity class; they inherit it from their persistent superclass).

Each additional argument to the tool must be one of the following:

- The full name of a persistent class.
- The `.class` file of a persistent class.
- A `.jdo` metadata file. The identity class of each class listed in the metadata will be generated.

The `.java` file generated for each identity class will be placed in the same directory as the parent class' `.java` file. If the generated file is overwriting an older file, the older file will be backed up to `<file-name>~`.

Note

The `.java` for the parent class must be in the `CLASSPATH`.

Example 3.1. Using the Application Identity Tool

There are many ways to invoke the `appidtool`.

```
java com.solarmetric.kodo.enhance.AppIdTool com.solarmetric.examples.Person
java com.solarmetric.kodo.enhance.AppIdTool -properties myapp.properties package.jdo
appidtool -licenseKey xxx-yyy-zzz com/solarmetric/examples/*.jdo
appidtool com.solarmetric.examples.Person Employee.class com/solarmetric/examples/Project.jdo
```

3.2. Enhancement

As discussed in the JDO Overview, JDO uses a process called *enhancement* to prepare persistent classes for management by the JDO runtime. The Kodo JDO enhancer is a command-line tool that can be invoked via the included `jdoc` script or via its java class, `com.solarmetric.kodo.enhance.JDOEnhancer`. The preferred method of enhancement is the `jdoc` script, which also validates the class metadata and points out any obvious inconsistencies.

The enhancer accepts the standard set of command-line arguments defined by the configuration framework. Like the `appidtool`, each additional argument to the enhancer must be either the full name of a persistent class, the `.class` file of a persistent class, or a `.jdo` metadata file listing one or more persistent classes.

You can run the enhancer over classes that have already been enhanced, in which case it will not further modify the class. You can also run it over classes that are not persistence-capable, in which case it will treat the class as persistence-aware.

Note that the enhancement process for subclasses introduces dependencies on the persistent parent class being enhanced. This is normally not problematic; however, when running the enhancer multiple times over a subclass whose parent class is not yet enhanced, class loading errors can occur. In the event of a class load error, simply re-compile and re-enhance the offending classes.

Example 3.2. Using the Kodo JDO Enhancer

The enhancer is used like the `appidtool`.

```
java com.solarmetric.kodo.enhance.JDOEnhancer com.solarmetric.examples.Person
java com.solarmetric.kodo.enhance.JDOEnhancer -properties myapp.properties package.jdo
jdoc -licenseKey xxx-yyy-zzz com/solarmetric/examples/*.jdo
jdoc com.solarmetric.examples.Person Employee.class com/solarmetric/examples/Project.jdo
```

3.3. Auto-Generating Classes from a Schema

Kodo JDO now includes a preview release of our upcoming reverse mapping tool. The reverse mapping tool generates persistent class definitions, complete with JDO metadata and Kodo mapping extensions, from an existing database schema. The tool is currently in its beta stages of development; bug reports are very much appreciated. The reverse mapping tool has been tested with the following databases:

- Hypersonic SQL 7.1
- SQLServer (MS Beta 2 JDBC driver)
- Sybase
- Oracle (9.0.1 JDBC driver)
- DB2
- Postgres (7.3 Beta 3 JDBC driver)

To use the reverse mapping tool, follow the steps below:

1. Set the `com.solarmetric.kodo.impl.jdbc.Schemas` configuration property to a comma-separated list of the names of the target database schemas. If you do not use explicit database schemas, you can skip this step (in this case Kodo JDO will attempt to filter out system tables on its own; for example under Oracle it

filters all tables with a schema of MDSYS, SYS, or SYSTEM).

Note that some databases require schema names to be in a certain case. If you are using Oracle, for example, your schema names must be in all upper case, regardless of what case you used when creating the schema.

2. Use the R&D schema generator to export your current schema to an XML schema file. The R&D schema generator can be run via the included `rd-schemagen` script or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.schema.SchemaGenerator`. For example:

```
rd-schemagen -properties myprops.properties -file schema.xml
```

You can limit the schemas and/or tables the schema generator looks at by listing the target schemas and tables on the command line. To target a schema, list its name. To target a table, list its full name in the form `<schema-name>.<table-name>`. If a target table does not have a schema or you do not know its schema, list its name as `.<table-name>`. The example below will generate XML for the entire BUSOBSJS schema, the ADDRESS table in the GENERAL schema, and the SYSTEM_INFO table, regardless of what schema it is in (or it may not have a schema).

```
rd-schemagen -properties myprops.properties -file schema.xml BUSOBSJS
GENERAL.ADDRESS .SYSTEM_INFO
```

3. Examine the generated schema file. JDBC drivers often provide incomplete or faulty metadata, in which case the file will not exactly match the actual schema. Alter the XML file to match the true schema. The DTD for the schema file is given here.

After fixing any errors in the schema file, modify the XML to include foreign keys between all related tables. The schema generator will have automatically detected existing foreign key constraints; many schemas, however, do not employ database foreign keys for every table relation. By manually adding any missing foreign keys, you will give the reverse mapping tool the information it needs to reflect the proper relations between the persistent classes it creates.

4. Run the reverse mapping tool on the finished schema file. (If you do not supply the schema file to reverse map, the tool will run directly against the schema in the database). The tool can be run via the included `rd-reversemappingtool` script, or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.meta.ReverseMappingTool`. In addition to the standard configuration flags accepted by all Kodo JDO tools, the reverse mapping tool recognizes the following command line flags:

- `-package <package name>`: The package name of the generated classes. If no package name is given, the generated code will not contain package declarations.
- `-file <output directory>`: The path to the directory to output all generated code and metadata to. Defaults to the current directory.
- `-useSchemaName <true / false>`: Set this flag to true to include the schema as well as table name in the name of each generated class. This can be useful when dealing with multiple schemas with same-named tables.
- `-useForeignKeyName <true / false>`: Set this flag to true if you would like field names for relations to be based on the database foreign key name. By default, relation field names are derived from the name of the related class.
- `-inheritance <true / false>`: Set this flag to false if you do not want the tool to create inheritance relationships based on foreign key information. Any foreign keys that would have indicated inheritance will be translated as 1-1 relationships instead.

- `-nullableAsObject <true / false>` : By default, all non-foreign key columns are mapped to primitives. Set this flag to true to generate primitive wrapper fields instead for columns that allow null values.
- `-primaryKeyOnJoin <true / false>` : The standard reverse mapping tool behavior is to map all tables with primary keys to persistent classes. If your schema has primary keys on many-many join tables as well, set this flag to true to avoid creating classes for those tables.
- `-metadata <package / class>`: Whether to write a single package-level JDO metadata file, or to write a JDO metadata file per generated class. Defaults to package.

Note

If you are using Kodo's JBuilder integration features, make sure to specify the `-metadata class` flag to write a separate JDO metadata file per class.

Standard usage example:

```
rd-reversemappingtool -properties myprops.properties -package com.xyz.jdo schema.xml
```

Running the tool will generate `.java` files for each generated class, package-level or per-class `.jdo` files, depending on the `-metadata` flag, a `<package-name>.schema` file, and a `<package-name>.mapping` file. The schema file is there to show you what portion of the schema was mapped; it serves no other purpose and can be deleted if desired. The mapping file contains the O/R mapping information for the generated classes. Mapping files like these will be offered as an optional alternative to O/R metadata extensions in a future version of Kodo JDO. For now, you must import the mapping information into Kodo JDO metadata extensions.

5. To import the generated O/R mapping information into metadata extensions, run the R&D import tool on the mapping file. The tool can be invoked via the included `rd-importtool` script or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.meta.compat.ImportTool`. Make sure to compile the generated classes before the import:

```
javac *.java
rd-importtool -properties myprops.properties jdo.mapping
```

You can now delete the mapping file if desired.

6. Examine the generated classes and metadata, and modify them as necessary.

3.3.1. Schema File DTD

```
<!ELEMENT schemas (schema)+>
<!ELEMENT schema (table)*>
<!ATTLIST schema name CDATA #IMPLIED>
<!ELEMENT table (column|index|pk|fk)+>
<!ATTLIST table name CDATA #REQUIRED>
<!ELEMENT column EMPTY>
<!ATTLIST column name CDATA #REQUIRED>
<!ATTLIST column type (array | bigint | binary | bit | blob | char | clob | date | decimal | distinct | float | int | java | long | short | string | text | time | timestamp | tinyint | varchar | varbinary) #REQUIRED>
<!ATTLIST column not-null (true|false) "false">
<!ATTLIST column default CDATA #IMPLIED>
<!ATTLIST column size CDATA #IMPLIED>
<!ATTLIST column decimal-digits CDATA #IMPLIED>
```

```

<!-- the 'column' attribute of indexes, pks, and fks can be used -->
<!-- when the element has only one column (or for foreign keys, -->
<!-- only one local column); in these cases the on/join child -->
<!-- elements can be omitted -->
<!ELEMENT index (on)*>
<!ATTLIST index name CDATA #REQUIRED>
<!ATTLIST index column CDATA #IMPLIED>
<!ATTLIST index unique (true|false) "false">
<!ELEMENT pk (on)*>
<!ATTLIST pk name CDATA #IMPLIED>
<!ATTLIST pk column CDATA #IMPLIED>
<!ELEMENT on EMPTY>
<!ATTLIST on column CDATA #REQUIRED>
<!ELEMENT fk (join)*>
<!ATTLIST fk name CDATA #IMPLIED>
<!ATTLIST fk to-table CDATA #REQUIRED>
<!ATTLIST fk column CDATA #IMPLIED>
<!ATTLIST fk delete-action (cascade|default|exception|none|null) "none">
<!ELEMENT join EMPTY>
<!ATTLIST join column CDATA #REQUIRED>
<!ATTLIST join to-column CDATA #REQUIRED>

```

3.4. Smart Proxies

Kodo JDO takes advantage of the guaranteed uniqueness of `java.util.Set` elements and `java.util.Map` keys to create *smart* proxies for your persistent set and map fields. Most proxies only track whether or not they have been modified. Smart proxies, however, keep a record of which elements have been added and removed. This record enables the Kodo JDO runtime to make more efficient database updates on these fields.

When designing your persistent classes, keep in mind that you can optimize for Kodo JDO by using fields of type `java.util.Set`, `java.util.TreeSet`, and `java.util.HashSet` for your collections whenever possible. You can also design your own smart proxies to further optimize Kodo JDO for your usage patterns. See the section on custom proxies for details.

Chapter 4. Metadata

Kodo JDO uses standard JDO metadata documents, which are covered in the JDO Overview. Kodo JDO also takes advantage of metadata's built-in extension mechanism to allow you to specify additional information in two categories:

- *Dependency information.* Dependency-related extensions are used to declare related objects that should be automatically deleted when the parent object is deleted.

Object-relational mapping information. Using object-relational mapping extensions, you can customize your objects' schema, or map persistent classes to an existing schema.

All metadata extensions are optional; Kodo JDO will define its own schema mapping and dependencies by default. If you do choose to specify metadata `extension` elements, they must have a `vendor-name` of `kodo`. The next sections present a list of the available extensions in each category.

4.1. Dependency Extensions

Marking a field as dependent means that when the owning object is deleted, the related object(s) stored in the field will be deleted as well. This process is recursive, so dependent objects can have dependent fields themselves as well. Dependent fields are analyzed during JDO transaction commit.

All dependency-related extensions are specified at the field level (i.e. as direct child elements to the `field` elements they apply to). They all take a value of `true` or `false`:

- `dependent`: Setting a value of `true` for this extension indicates that the persistent object stored in this *one-to-one* relation field should be deleted when the parent object is deleted.
- `element-dependent`: This extension is like the `dependent` extension, but applies to the element values of collection fields. Use this extension for *one-to-many* or *many-to-many* relations where the related objects should be deleted along with the owning object.
- `value-dependent`: This extension is equivalent to the `element-dependent` extension above, but is used for map values rather than collection elements.
- `key-dependent`: This extension applies to map keys. (*Note: first class map keys are not yet supported.*)

4.2. Class-level Object-Relational Mapping Extensions

The following keys are recognized for `extension` elements that are direct children of `class` elements in the metadata document:

- `table`: This extension specifies the table used to store the primary data for the class. If not specified, Kodo will auto-generate a table name based on the name of the class. This attribute is also used in multi-table inheritance mappings
- `pk-column`: This extension is only for classes using datastore identity. It specifies the primary key column for the table in which the class is held. This column must be of a numeric type and must *not* be mapped to any fields of the class. If the `pk-column` extension is not specified, Kodo will add its own primary key column, usually named `JDOIDX`.
- `lock-column`: This extension specifies the column used to record the version number of objects. Versioning

is used to detect concurrent modification of objects during optimistic transactions. The given column must be of a numeric type and must *not* be mapped to any fields of the class. If the extension is not present, Kodo JDO will add its own lock column, usually named `JDOLOCKX`. You can prevent the creation of a lock column by specifying a value of `none`. In this case, concurrent modification violations will not be detected.

- `class-column`: This column stores the class name of the object represented by each table row. The column must be a string type, and must be large enough to hold the full class name of any persistent class mapped to the table. It must *not* be mapped to any fields of the class. If the extension is not present, Kodo JDO will add its own class column, usually named `JDOCLASSX`. If the table's corresponding persistent class has no persistent subclasses and you do not want a column to be generated, specify a value of `none`.
- `subclass-provider`: This optional extension stores the name of the `SubclassProvider` implementation to use for this class. This setting overrides the `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass` configuration property.

This should only be set in the least-derived type in an inheritance tree. Values for this extension in derived types are silently ignored.

- `subclass-indicator-value`: When using the `IntegerSubclassProvider` subclass provider, this required extension must be set to the integer value that the provider should use to identify this class. No two classes in the same class hierarchy can use the same integer value.
- `custom-mapping`: This optional extension stores the name of the `ClassMapping` subclass to use for this class. This setting overrides the `com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass` configuration property.

This should only be set in the least-derived type in an inheritance tree. Values for this extension in derived types are silently ignored.

- `can-cache`: This optional extension designates whether or not a given class should be included in the `PersistenceManagerFactory` cache, if any. Setting it to `false` will exclude the class associated with this extension. Defaults to `true`.

4.3. Field-level Object-Relational Mapping Extensions

The following keys are recognized for extension elements that are direct children of field elements in the metadata document:

- `dependent`: Setting a value of `true` for this extension indicates that the persistent object stored in this field should be deleted when the parent object is deleted. The deletion of dependent field values occurs during the commit process.
- `element-dependent`: This extension is equivalent to the `dependent` extension above, but applies to the element values of collection fields. Use this extension for *one-to-many* or *many-to-many* relations where the related objects should be deleted along with the owning object.
- `value-dependent`: This extension is equivalent to the `dependent` extension above, but applies to first class map values.
- `key-dependent`: This extension is equivalent to the `dependent` extension above, but applies to first class map keys (*note: first class map keys are not yet supported*).
- `inverse`: Specifies the field name of the other side of a two-sided relationship between objects. It is safe to leave out this extension, but the relational schema can be more efficient when the inverse is known. In object-relational mapping terms, inverses are used to detect *2-sided one-to-ones*, *one-to-many* and *2-sided many-to-many* mappings.

- `blob`: Takes a value of `true` or `false`. This extension can be used to explicitly mark fields that should be stored as serialized BLOB values. For example, Kodo JDO would normally store a field of type `byte[]` using a secondary table with a row for each byte value in the array; this type of storage allows for queries on the array. If the `byte[]` is large or contains information that should not be query-able, it may be more efficient to store it as a BLOB column of the main table. Fields that Kodo JDO cannot normally persist, such as user-defined interface fields or fields of type `java.lang.Object` are defaulted to use BLOB mappings. In these cases this extension is not necessary, though it is still allowed.
- `column-length`: String fields can specify the maximum length of the String that will be stored in the database column. The default maximum is 255 characters. Use a value of -1 to indicate that there should be no length limit on the String. In this case, the column will be created as a CLOB, or the correct database equivalent. Currently, this extension is used only when creating a schema using the `schematool`. In the future, this extension may be also used for performing runtime data validations.
- `column-index`: This extension takes a value of `true` or `false` to specify whether the column holding the data for the field should be indexed.
- `ordered`: This extension accepts a value of `true` or `false`. By default, databases do not guarantee the order in which results are read. Thus, collection and array field elements may be retrieved from the database in a different order from the one in which they were stored. A value of `true` for this key, however, will ensure that the order of the elements in this field is retained during database reads. This flag is useful only for collection and array fields that represent a one-sided relationship, such as a collection of simple values or a 1-sided many-to-many relation. It cannot be used for one-to-many relations or shared many-to-many relations. See the `order-column` extension for more information.
- `table`: Some mappings do not store their data in the class' primary table. This extension specifies the table where the data for this field is stored. This attribute is also used in multi-table inheritance mappings.

For collection, array, and map fields, this is the secondary or cross-reference table used to hold the field value. Normally, Kodo JDO auto-generates secondary tables based on the class and field name. Note that this extension should not be used for one-to-many collections, since there is no secondary table for such collections.

- `data-column`: This extension specifies the column in which the data for this field is stored. For simple fields, this column must belong to the *primary table* of the class (i.e., the table in which the primary key value is stored). For collection and map fields, this column can belong to a secondary table, as long as an additional `ref` column is provided to link the secondary table to the primary (see below). If this field represents a relation to another persistent object, the named column should hold the primary key value of the related instance. Note that this extension is ignored if the class to which this field refers uses application identity. The next section covers application identity extensions.
- `key-column`: This extension is applicable only to map fields. It specifies the column of the secondary table that holds the key values.
- `ref-column`: This extension is only for a collection or map field whose data column exists in a secondary table. The `ref-column` must be in the same table as the data column, and must contain the primary key value of the owning instance. This allows a SQL join to be performed linking this `ref` column with the primary key column of the primary table. Note that this extension is ignored if the class to which this field refers uses application identity. The next section covers application identity extensions.
- `order-column`: This extension is applicable only to collection fields that also have the `ordered` extension. It specifies the column of the secondary table used to hold ordering information. This column must be of a numeric type. If the `ordered` extension is given but this extension is omitted, Kodo JDO will create its own order column, usually named `JDOORDERX`.
- `read-only`: This extension is used for inverse one-to-one relations and collection fields that represent a shared many-to-many mapping.

In inverse one-to-one relations, this meta data flag marks which class does *not* own the field in its table, and thus

has a read-only view of the relational data. If neither side of such a relation is marked read-only, Kodo will automatically choose one side as such.

In many-to-many collection field mappings, one class can declare its field to be read-only in order to avoid duplicate inserts into the shared table. The value of this extension should be either `true` or `false`. Note that Kodo JDO typically makes one side of the relation read-only automatically; this extension is never strictly needed.

- `custom-mapping`: This optional extension stores the name of the `FieldMapping` subclass to use for this field.

4.4. Extensions Under Application Identity

Classes that use application identity require slightly different extensions than those enumerated above. First, no `pk-column` extension is ever needed, since the primary key columns can be determined from the columns for the primary key fields. Second, because application identity classes may use multiple primary key columns, all extensions associated with primary keys must be prefixed with the name of the corresponding field. This applies to all `ref-column` extensions, and to `data-column` extensions for one-to-one and many-to-many relation fields. A concrete example using application identity is presented in the next section.

4.5. Examples

The examples below show extensions being used to map persistent classes to an existing schema. Remember, however, that you are free to let Kodo auto-generate the schema. In this case, you might still use a few extensions, but only to specify inverses for one-to-many relations, turn on ordering in collection or array fields, and customize the names of a few tables and columns in the event of naming conflicts.

Example 4.1. Mapping Classes to an Existing Schema

For the purpose of this example, assume the following tables exist:

PERSON

| PK | NAME | SSNUM | MGR |
|-----|-------|-------------|------|
| 1 | john | 123-45-6789 | 3 |
| 2 | fred | 987-65-4321 | 3 |
| 3 | alice | 111-22-3333 | NULL |
| ... | | | |

PROJECT

| PK | START_DATE | END_DATE | CODE_NAME |
|-----|------------|------------|------------|
| 1 | 10 25 2000 | NULL | 2.0a1 |
| 2 | 01 10 2001 | 01 31 2001 | TestPhase1 |
| ... | | | |

XREF

| PERSON | PROJECT |
|--------|---------|
| 1 | 1 |
| 2 | 1 |
| 2 | 2 |

```
|=====|
```

The representation of this data as java classes might look something like:

```
package com.solarmetric.examples;

...

public class Person
{
    private String      name;
    private String      social;
    private Person      manager;
    private Collection   managedPeople;
    private Collection   projects;

    ...
}

public class Project
{
    private String      codeName;
    private Date        start;
    private Date        end;
    private List         people;

    ...
}
```

Given the above classes and tables, the metadata to map them together could be defined as follows:

```
<?xml version="1.0"?>
<jdo>
  <package name="com.solarmetric.examples">
    <class name="Person">
      <extension vendor-name="kodo" key="table" value="PERSON"/>
      <extension vendor-name="kodo" key="pk-column" value="PK"/>
      <extension vendor-name="kodo" key="lock-column" value="none"/>
      <extension vendor-name="kodo" key="class-column" value="none"/>
      <field name="name">
        <extension vendor-name="kodo" key="data-column" value="NAME"/>
      </field>
      <field name="social">
        <extension vendor-name="kodo" key="data-column" value="SSNUM"/>
      </field>
      <field name="manager">
        <extension vendor-name="kodo" key="data-column" value="MGR"/>
      </field>

      <!--
        one-to-many mappings never take schema extensions; all the
        needed information is available from the related class and its
        inverse field
      -->
      <field name="managedPeople">
        <collection element-type="Person"/>
        <extension vendor-name="kodo" key="inverse" value="manager"/>
      </field>
      <field name="projects">
        <collection element-type="Project"/>
        <extension vendor-name="kodo" key="inverse" value="people"/>
        <extension vendor-name="kodo" key="table" value="XREF"/>
        <extension vendor-name="kodo" key="data-column" value="PROJECT"/>
        <extension vendor-name="kodo" key="ref-column" value="PERSON"/>
      </field>
    </class>
  </package>
</jdo>
```

```

</class>
<class name="Project">
  <extension vendor-name="kodo" key="table" value="PROJECT"/>
  <extension vendor-name="kodo" key="pk-column" value="PK"/>
  <extension vendor-name="kodo" key="lock-column" value="none"/>
  <extension vendor-name="kodo" key="class-column" value="none"/>
  <field name="codeName">
    <extension vendor-name="kodo" key="data-column" value="CODE_NAME"/>
  </field>
  <field name="start">
    <extension vendor-name="kodo" key="data-column" value="START_DATE"/>
  </field>
  <field name="end">
    <extension vendor-name="kodo" key="data-column" value="END_DATE"/>
  </field>
  <field name="people">
    <collection element-type="Person"/>
    <extension vendor-name="kodo" key="inverse" value="projects"/>
    <extension vendor-name="kodo" key="table" value="XREF"/>
    <extension vendor-name="kodo" key="data-column" value="PERSON"/>
    <extension vendor-name="kodo" key="ref-column" value="PROJECT"/>
  </field>
</class>
</package>
</jdo>

```

Example 4.2. Extensions Under Application Identity

Now we will modify the example a bit for application identity. The new schema is:

```

PERSON
=====
| NAME (PK) | SSNUM (PK) | MGR_NAME | MGR_SSNUM |
|-----|-----|-----|-----|
| john      | 123-45-6789 | alice    | 111-22-3333 |
| fred      | 987-65-4321 | alice    | 111-22-3333 |
| alice     | 111-22-3333 | NULL     | NULL        |
|           | ...         |          |             |
|=====|
PROJECT
=====
| START_DATE | END_DATE | CODE_NAME (PK) |
|-----|-----|-----|
| 10 25 2000 | NULL    | 2.0a1          |
| 01 10 2001 | 01 31 2001 | TestPhase1     |
|           | ...         |                 |
|=====|
XREF
=====
| PERSON_NAME | PERSON_SSNUM | PROJECT |
|-----|-----|-----|
| john      | 123-45-6789 | 2.0a1   |
| fred      | 987-65-4321 | 2.0a1   |
| alice     | 111-22-3333 | TestPhase1 |
|=====|

```

The classes remain unchanged. Application identity classes would have to be introduced for `Person` and `Project`, but those will not be shown here. To review, the classes representing the above tables are:

```
package com.solarmetric.examples;
```

```

...
public class Person
{
    private String      name;
    private String      social;
    private Person      manager;
    private Collection  managedPeople;
    private Collection  projects;

    ...
}

public class Project
{
    private String  codeName;
    private Date    start;
    private Date    end;
    private List    people;

    ...
}

```

Given the above classes and tables, the metadata to map them together could now be defined as follows:

```

<?xml version="1.0"?>
<jdo>
  <package name="com.solarmetric.examples">
    <class name="Person" objectid-class="...">
      <extension vendor-name="kodo" key="table" value="PERSON"/>
      <extension vendor-name="kodo" key="lock-column" value="none"/>
      <extension vendor-name="kodo" key="class-column" value="none"/>
      <field name="name" primary-key="true">
        <extension vendor-name="kodo" key="data-column" value="NAME"/>
      </field>
      <field name="social" primary-key="true">
        <extension vendor-name="kodo" key="data-column" value="SSNUM"/>
      </field>
      <field name="manager">
        <extension vendor-name="kodo" key="name-data-column" value="MGR_NAME"/>
        <extension vendor-name="kodo" key="social-data-column" value="MGR_SSNUM"/>
      </field>

      <!--
        one-to-many mappings never take schema extensions; all the
        needed information is available from the related class and its
        inverse field
      -->
      <field name="managedPeople">
        <collection element-type="Person"/>
        <extension vendor-name="kodo" key="inverse" value="manager"/>
      </field>
      <field name="projects">
        <collection element-type="Project"/>
        <extension vendor-name="kodo" key="inverse" value="people"/>
        <extension vendor-name="kodo" key="table" value="XREF"/>
        <extension vendor-name="kodo" key="codeName-data-column" value="PROJECT"/>
        <extension vendor-name="kodo" key="name-ref-column" value="PERSON_NAME"/>
        <extension vendor-name="kodo" key="social-ref-column" value="PERSON_SSNUM"/>
      </field>
    </class>
    <class name="Project" objectid-class="...">
      <extension vendor-name="kodo" key="table" value="PROJECT"/>
      <extension vendor-name="kodo" key="lock-column" value="none"/>
      <extension vendor-name="kodo" key="class-column" value="none"/>
      <field name="codeName" primary-key="true">
        <extension vendor-name="kodo" key="data-column" value="CODE_NAME"/>

```

```

</field>
<field name="start">
  <extension vendor-name="kodo" key="data-column" value="START_DATE"/>
</field>
<field name="end">
  <extension vendor-name="kodo" key="data-column" value="END_DATE"/>
</field>
<field name="people">
  <collection element-type="Person"/>
  <extension vendor-name="kodo" key="inverse" value="projects"/>
  <extension vendor-name="kodo" key="table" value="XREF"/>
  <extension vendor-name="kodo" key="name-data-column" value="PERSON_NAME"/>
  <extension vendor-name="kodo" key="social-data-column" value="PERSON_SSNUM"/>
  <extension vendor-name="kodo" key="codeName-ref-column" value="PROJECT"/>
</field>
</class>
</package>
</jdo>

```

4.6. Multi-table Inheritance Mapping

By default, Kodo JDO maps all fields in a class hierarchy into the table defined by the least-derived type in that class hierarchy. This is often the best class mapping strategy to use, as it reduces the number of joins and separate statements necessary, reducing load on the database and increasing database performance. However, there are many situations in which this mapping is not ideal. Inheritance hierarchies with lots of fields might be better broken up into several tables in order to reduce overall table size, or to reduce the amount of data transmitted for a given select. Existing database schema constraints may require the use of multiple tables per hierarchy, or even multiple tables per individual class.

It is possible to control which tables a given class maps to on both per-class and per-field levels.

To designate the table that all fields declared in a class should map to, use the `table` metadata extension on the `class` element in the JDO metadata. If this extension is set, then Kodo will by default map all fields in this class into that table. Otherwise, the behavior is controlled by the `com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping` configuration property. Configuration is covered here. If this property is set to `true` (the default value), then the fields will by default be mapped into the table in use by the superclass of the class. Otherwise, a new table name will be generated based on the class name, and a table with that name will be used by default for all fields in the class.

To designate a table that a particular field should map to, use the `table` metadata extension on a `field` element in the JDO metadata. If this extension is set, then Kodo will map the given field's primary table data into the specified table. Otherwise, the field's primary table data will be mapped into the table in use by the owning class.

When presented with a class hierarchy that spans multiple tables, Kodo requires that all tables in the hierarchy share the same number of primary key fields and that the primary key column names be identical.

When loading a class that spans multiple tables and for which the entire class hierarchy maps to the same set of tables, Kodo performs a single select statement that joins the related tables on the primary keys. When loading a class that maps to additional tables not mapped to by the class' superclass, Kodo performs multiple select statements.

4.7. Generating Default JDO Metadata

Kodo JDO now includes a preview release of our upcoming metadata tool. The metadata tool uses reflection to generate default JDO metadata for your persistent classes. It cannot fill in information that is not available from the class definition itself, such as the element type of collections or the primary key fields of a class using application identity. It does, however, provide a good starting point from which to build up your metadata.

The metadata tool recognizes JDO's extensive system of defaults, so fields that are persistent by default will not be included in the generated XML document. The only exception to this rule is for collection and map fields: the tool adds these fields to the metadata and sets their `element-type`, `key-type`, and `value-type` attributes to `Object` as a reminder to you to provide this information.

The metadata tool can be run via the included `rd-metadatatool` script, or through its Java class, `com.solarmetric.rd.kodo.meta.JDOMetaDataTool`. In addition to the standard configuration flags accepted by all Kodo JDO tools, the reverse mapping tool recognizes the following command line flags:

- `-file <metadata file>`: The name of the metadata file to generate. If this argument is not supplied, the tool will print the generated metadata to stdout.

Each additional argument to the tool should be the class name, .class file, or .java file of a class to generate metadata for. Each class must be compiled.

Standard usage example:

```
rd-metadatatool -properties myprops.properties -file mypackage/mypackage.jdo
mypackage/*.java
```

Chapter 5. JDBC Configuration

Kodo JDO uses a relational database for object persistence. It communicates with the database using the Java DataBase Connectivity (JDBC) APIs. The following standard JDO properties are used to configure JDBC connectivity:

- `javax.jdo.option.ConnectionUserName`
- `javax.jdo.option.ConnectionPassword`
- `javax.jdo.option.ConnectionURL`
- `javax.jdo.option.ConnectionDriverName`

Refer to the configuration framework for details on these and other properties.

5.1. Supported Databases

Kodo JDO can take advantage of any JDBC 1.x compliant driver, making almost any major database a candidate for use. See our officially supported database list for more information.

If your database is not officially supported, you can add support for it by implementing your own `DBDictionary`. This is typically accomplished by extending the concrete `GenericDictionary` class. You can then plug your dictionary into Kodo JDO using the `com.solarmetric.kodo.impl.jdbc.DictionaryClass` and `com.solarmetric.kodo.impl.jdbc.DictionaryProperties` configuration properties. These properties are described in the chapter examining the configuration framework.

Example 5.1. Configuring custom dictionary properties

```
com.solarmetric.kodo.impl.jdbc.DictionaryClass: \
com.solarmetric.kodo.impl.jdbc.schema.dict.HSQLDictionary

com.solarmetric.kodo.impl.jdbc.DictionaryProperties: \
SchemaName=MySchema SupportsSelectForUpdate=false
```

5.2. Accessing Multiple Databases

Through the properties above, each `PersistenceManagerFactory` can be configured to access a different database. If your application accesses multiple databases, we recommend that you maintain a separate properties file for each one. This will allow you to easily load the appropriate resource for each database at runtime, and to given the correct file to Kodo JDO's command-line tools during development.

5.3. Connection Management

By default, Kodo performs database connection pooling with its own `javax.sql.DataSource` implementation. The numbers provided in the `javax.jdo.option.MinPool` and `javax.jdo.option.MaxPool` configuration properties specify the number of connections that the pool should maintain. You can also manage connections yourself by setting the `ConnectionFactory` of the `PersistenceManagerFactory` to your own `javax.sql.DataSource`. Alternatively, you can set the `javax.jdo.option.ConnectionFactoryName` configuration property to the JNDI location of a bound

```
javax.sql.DataSource.
```

If you would prefer to use a `javax.sql.DataSource` other than Kodo's built-in pooling implementation, you can set the `DataSource`'s class name in the `javax.jdo.option.ConnectionDriverName` property, and specify the bean properties in the `com.solarmetric.kodo.ConnectionProperties` property. In these cases, the `javax.jdo.option.ConnectionURL` property will be unused.

Example 5.2. Properties for using a custom DataSource

```
javax.jdo.option.ConnectionDriverName=oracle.jdbc.pool.OracleDataSource
com.solarmetric.kodo.ConnectionProperties=PortNumber=1521 \
                                         ServerName=saturn \
                                         DatabaseName=solarsid \
                                         DriverType=thin
javax.jdo.option.ConnectionUserName=jdotestuser
javax.jdo.option.ConnectionPassword=jdotestpassword
```

Important

Kodo's built-in `DataSource` implementation performs advanced connection pooling and caching of `PreparedStatement`. Using a third-party `DataSource` that does not provide these features may result in sub-optimal Kodo performance.

5.4. Large Result Sets

When using a JDBC driver that supports version 2.0 or higher of the JDBC specification, and that supports `ResultSet`s of type `TYPE_SCROLL_INSENSITIVE`, collections returned from `Query.execute` methods will load rows from the database only when they are needed. This on-demand loading is also applied to `javax.jdo.Extents`.

The on-demand instantiation of elements of the `ResultSet` allows efficient handling of potentially very large result sets. Kodo JDO further conserves memory by using soft references to hold objects as they are instantiated during iteration. If the JVM begins to run low on memory, these objects will be garbage collected. Thus even query results or extents containing millions of objects can be fully iterated.

If an on-demand collection is ever completely instantiated, its soft references will be converted to hard references, and any database resources it is using will be freed. It becomes a standard collection of objects.

To facilitate users who require random access to query results, Kodo JDO always returns an implementation of `java.util.List` from calls to `Query.execute`. Remember, though, that other JDO implementations might choose to only implement the `java.util.Collection` interface in their query result objects. Also, note that unless ordering is specified in your query, there is no guarantee that multiple executions of the same query will return their results in the same order.

Example 5.3. Using Random Access Query Results in a Portable Fashion

```
// get start and end indexes of results to display from web request
int startIndex = Integer.parseInt (jspRequest.getParameter ("start"));
int endIndex = Integer.parseInt (jspRequest.getParameter ("end"));

Extent extent = pm.getExtent (Product.class, true);
Query query = pm.newQuery (extent, "productName == \"Stereo\"");
```

```
// we want to ensure that we always order the results in the same way
query.setOrdering ("productCode ascending");

Collection results = (Collection) query.execute ();
List resultList;
if (results instanceof List)
    resultList = (List) results;
else
    // portable, but it will wind up instantiating all elements in
    // collection, which might be a huge number of objects
    resultList = new ArrayList (results);

// print info about each product in list range from "start" to "end"
for (int i = startIndex; i <= endIndex && i < resultList.size (); i++)
    out.print ("Stereo #" + i + ": "
        + ((Product) resultList.get (i)).getDescription ());
```

5.5. Schema Manipulation

Kodo JDO stores persistent objects in relational database tables. As you add, remove, or modify your persistent classes, these tables must be updated to reflect the current object model.

To facilitate this process, Kodo JDO provides the `schematool`. This command-line tool can create, refresh, or drop the relational schema for any persistent types, relieving you from database administration tasks. You can invoke the tool through the included `schematool` script or via its java class, `com.solarmetric.kodo.impl.jdbc.schema.SchemaTool`. It accepts the standard set of command-line arguments defined by the configuration framework. It also accepts the following flags:

- *-ignoreErrors <true/false>*: Whether the tool should ignore SQL errors that are thrown while it is manipulating the schema. This optional argument defaults to `false`.
- *-outfile <filename>*: If this optional argument is set, then the `schematool` will write SQL statements to the specified file rather than perform any modifications to the data store. Use the keyword `stdout` to print the SQL to standard output.
- *-action <add/refresh/drop>*: This flag is required. It tells the tool what action to perform, where the available actions are:
 - *add*: Creates the schema for the given persistent types, if it does not already exist. This action will add tables for new classes, secondary tables for new collection, array, and map fields, or add new columns to existing tables for new simple fields. If the schema is already up-to-date, the tool will detect this and will not perform any additional work. This action will never drop data.
 - *refresh*: Similar to *add*, but also detects columns that are no longer used and drops them from the schema (some databases do not support dropping columns).
 - *drop*: Drops all schema components used by the given persistent types. If a subclass is mapped to the same table as its parent class, and only the subclass is included in the list of types to drop, the parent class' table will not be destroyed. If possible, the tool will remove the subclass' columns from the table. Be careful when using this action!
- *-db <db name>*: This option is for users of the deprecated `system.prefs` configuration mechanism. Use it to specify the symbolic name of the database to connect to.

Any additional arguments to the `schematool` will be interpreted as persistent types whose schema should be modified. Just as with the `appidtool`, each of the arguments can be either a full class name, a `.class` file, or a `.jdo` file. If no classes are given, the action will be performed on all known persistent types.

Important

When acting on a persistent type, the `schematool` automatically extends the action to all known subclasses of the type.

Example 5.4. Using the Kodo JDO Schematool

Refresh the schema for all known persistent classes:

```
schematool -action refresh
```

Drop the schema for the `Company` and any types listed in the `package.jdo` file.

```
schematool -properties hsql.properties -action drop com.solarmetric.examples.Company ../package.jdo
```

Note

It is possible to bypass the `schematool` step by specifying the `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` Kodo property to `true`. This should never be used on a production database, since automatic schema migration can result in columns being dropped, and thus in data loss. See the Configuration Framework chapter for more details.

Chapter 6. Standard Features

Kodo JDO Standard Edition includes a number of extensions to the JDO specification and additional features that can be useful when developing an application with custom needs. We have already covered generating application identity classes, object-relational mapping extensions, including multi-table inheritance mapping, connection pooling, large result set support, and automatic schema manipulation. This chapter outlines additional important features of the standard edition.

6.1. Manipulating Datastore Identity Objects

Datastore identity objects are typically opaque. Kodo JDO does, however, allow you to manipulate these objects if you need to. All datastore identity objects in Kodo JDO are instances of the public `com.solarmetric.util.ObjectIds.Id` class. See its Javadoc for details.

6.2. ExtentImpl

Often, it is necessary to retrieve all items in an extent for processing of some sort. The `Extent` interface provides a mechanism for retrieving an `Iterator` that can be used to traverse the entire extent, but it does not provide a method for retrieving a collection of all the items in the extent.

Kodo JDO's `Extent` implementation provides a method that can be used to access the data in an extent via the `List` interface. See the `com.solarmetric.kodo.runtime.ExtentImpl` Javadoc for details.

6.3. PersistenceManager Extension

Some advanced users may want to use a custom `PersistenceManager` in place of Kodo JDO's `com.solarmetric.kodo.runtime.PersistenceManagerImpl`.

Kodo JDO permits simple extension of the `PersistenceManager` used by the `PersistenceManagerFactory`. This can be useful when custom behavior is desired, or when an application needs to receive notification when certain `PersistenceManager` methods are invoked.

To specify a subclass of `PersistenceManagerImpl`, use the `com.solarmetric.kodo.PersistenceManagerClass` property. The class specified must extend `com.solarmetric.kodo.runtime.PersistenceManagerImpl` -- the method will throw a `JDOUserException` if it does not. You can also setup your custom `PersistenceManager` through the `com.solarmetric.kodo.PersistenceManagerClass` and `com.solarmetric.kodo.PersistenceManagerProperties` properties, as described in the configuration framework.

6.4. Custom Proxies

At runtime, the values of all second class object fields in persistent and transactional objects are replaced with implementation-specific proxies. On modification, these proxies notify their owning instance that they have been changed, so that the appropriate updates can be made on the data store.

In Kodo JDO, proxies are managed through the `com.solarmetric.kodo.util.ProxyManager` interface. Kodo JDO includes a default `ProxyManager` implementation that will meet the needs of most users. For custom behavior, though, Kodo JDO allows you to define your own `ProxyManager`, and your own proxy classes. See the `com.solarmetric.util` package Javadoc for details on the interfaces involved, and the utility classes Kodo JDO provides to assist you.

You can plug your custom proxy manager into the Kodo JDO runtime through the `com.solarmetric.kodo.ProxyManagerClass` and `com.solarmetric.kodo.ProxyManagerProperties` configuration properties. The configuration framework is described here.

6.5. Access to SQL Connections

Kodo JDO provides two mechanisms for obtaining a `java.sql.Connection` object. This can be useful when direct access to the underlying data store is required.

The following code obtains the connection that is currently in use by a particular `PersistenceManager`. If there is no connection open to the data store for the given thread, then a new one is created and returned. If a data store transaction is in progress, then the connection returned will be transactionally consistent. See the Javadoc for `com.solarmetric.kodo.impl.jdbc.runtime.JDBCStoreManager` for more details.

Example 6.1. Obtaining a `java.sql.Connection` object from the `PersistenceManager`

```
PersistenceManagerFactory factory = ...; // obtain a PersistenceManagerFactory
PersistenceManagerImpl pm =
    (PersistenceManagerImpl) factory.getPersistenceManager ();
JDBCStoreManager storeManager = (JDBCStoreManager) pm.getStoreManager ();
Connection conn = storeManager.getConnection ();

// do stuff

storeManager.releaseConnection (conn);
```

The connection returned can safely be released regardless of the current transactional state, and regardless of whether or not the `PersistenceManager` is being closed. Releasing the connection does not close the connection unless appropriate.

Additionally, a connection that is in no way linked to the current `PersistenceManager` can be obtained using Kodo JDO:

```
PersistenceManagerFactory factory = ...;
DataSource dataSource = (DataSource) factory.getConnectionFactory ();
Connection conn = dataSource.getConnection ();
```

Note that for this example, it is up to you to do all cleaning up. Additionally, you may need to enter a username and password when obtaining a connection from the `DataSource`, depending on how you created the `PersistenceManagerFactory`.

6.6. `PersistenceManagerImpl.evictAll()` API extensions

Kodo JDO's `PersistenceManager` implementation includes two API extensions useful when reusing `PersistenceManager` instances. `PersistenceManagerImpl.evictAll(Class)` and `PersistenceManagerImpl.evictAll(Extent)` behave just like `PersistenceManager.evictAll()`, except that they operate only on instances that are members of the specified class or extent.

6.7. Custom Class Indicators

By default, Kodo JDO stores the full class name of inserted objects in the database, so that it can determine which type to instantiate when loading data from the database. However, existing databases often have a pre-existing mechanism for storing subclass indicator values. This might be based on a C++ class naming convention, or it might be a numeric indicator. Sometimes, subclass information might not be stored in a single column at all, but might only be determinable by analyzing other values in the result set.

Kodo JDO includes an optional subclass provider that stores integers in the database instead of the full classname. This can be considerably faster than the default provider. To use this alternate provider, set the `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass` configuration property to `com.solarmetric.kodo.impl.jdbc.ormapping.IntegerSubclassProvider`, and set the `subclass-indicator-value` metadata extension in all your classes to the appropriate integer value. Alternately, this can be performed on a class-by-class basis, by setting the `subclass-provider` metadata extension to `com.solarmetric.kodo.impl.jdbc.ormapping.IntegerSubclassProvider`.

A demonstration: these mappings are specified as demonstrated in the following example, in which rows representing `com.acme.object.Foo` have a value of 1 stored in the `JDOCLASSX` column, and rows representing `com.acme.object.Bar` have a value of 2:

Example 6.2. IntegerSubclassProvider example

```
<jdo>
  <package name="com.acme.object">
    <class name="Foo">
      <!-- specify that we should use this subclass provider instead
           of the default string-based provider for this class. This
           should only be set in the least-derived type in an
           inheritance hierarchy. -->
      <extension vendor-name="kodo" key="subclass-provider"
        value="com.solarmetric.kodo.impl.jdbc.ormapping.IntegerSubclassProvi

      <!-- set up the mapping for this class. This must be done for
           every class in the inheritance hierarchy, and all values
           must be unique. -->
      <extension vendor-name="kodo" key="subclass-indicator-value"
        value="1"/>
    </class>

    <class name="Bar" persistence-capable-superclass="Foo">
      <!-- set up the mapping for this class. This must be done for
           every class in the inheritance hierarchy, and all values
           must be unique. -->
      <extension vendor-name="kodo" key="subclass-indicator-value"
        value="2"/>
    </class>
  </package>
</jdo>
```

Kodo JDO also supports customization of this default behavior through the `SubclassProvider` interface and the `com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass` configuration property. Additionally, alternate subclass providers can be set on a per-class basis in the JDO metadata. See the [Class-level Object-Relational Mapping Extensions](#) documentation for more information on doing this.

Chapter 7. Enterprise Features

Kodo JDO Enterprise Edition includes a number of proprietary features targeted at the enterprise developer. This chapter outlines these features.

7.1. EEPersistenceManager object listeners

Often, it is desirable to be able to do extra processing on a global basis when a change is made to an object. The JDO specification provides the `javax.jdo.InstanceCallbacks` interface to this end. However, if you wish to perform extra logic regardless of the type of the object, it is not ideal to require developers to make a callback to some global change listener. For example, if your application wished to log all deletes, you could use this callback mechanism to perform this logging without requiring that all your persistence-capable objects implement the `jdoPreDelete()` method.

The `EEPersistenceManager` uses the `com.solarmetric.kodo.ee.ObjectEventListener` interface to permit this type of global callback behavior. See the JavaDoc included with this documentation for more information on this interface.

7.2. EEPersistenceManager transaction listeners

Similarly, a developer may want to receive notification of transaction events. The `EEPersistenceManager` uses the `com.solarmetric.kodo.ee.TransactionEventListener` for this purpose. See the JavaDoc included with this documentation for more information on this interface.

7.3. Datastore Cache

7.3.1. Overview of Kodo JDO Datastore Caching

Kodo JDO includes support for an optional datastore cache that operates at the `PersistenceManagerFactory` level. This cache is designed to significantly increase performance while remaining in full compliance with the JDO standard. This means that turning on the caching option can transparently increase the performance of your application, with no changes to your code base.

Kodo JDO's datastore cache is not related to the `PersistenceManager` cache dictated by the JDO specification. The JDO specification mandates behavior for the `PersistenceManager` cache aimed at guaranteeing transaction isolation when operating on persistent objects. Kodo JDO's datastore cache is designed to provide significant performance increases over cacheless operation, while guaranteeing that all JDO behavior will be identical in both cache-enabled and cacheless operation.

When enabled, the cache is checked before making a trip to the data store. Data is stored in the cache when objects are committed and when persistent objects are loaded from the datastore.

There are currently two versions of the datastore cache: a single JVM version and a distributed version. Both versions are bundled with Kodo JDO Enterprise Edition. They are available as optional plug-ins for Kodo JDO Standard Edition.

The single JVM version maintains and shares a data cache across all `PersistenceManager` instances obtained from a particular `PersistenceManagerFactory`. This version is not appropriate for use in a distributed environment, as caches in different JVMs or created from different `PersistenceManagerFactory` objects will not be synchronized.

The distributed version communicates cache invalidation information to other JVMs via JMS, TCP, or UDP, or by

using a Tangosol Coherence cache. See the descriptions of the different distributed caches below for more information.

7.3.2. Kodo JDO Cache Usage

To enable the basic single-`PersistenceManagerFactory` cache, set the `com.solarmetric.kodo.DataCacheClass` property to `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`.

To enable the distributed `PersistenceManagerFactory` cache, set the `com.solarmetric.kodo.DataCacheClass` property to `com.solarmetric.kodo.runtime.datacache.plugins.UDPCache`.

The default cache implementations maintain a least-recently-used map of object IDs to cache data. By default, 1000 elements are kept in cache. This can be adjusted by setting `com.solarmetric.kodo.DataCacheProperties` appropriately -- see below for an example. Removed objects are moved to a soft reference map, so they may stick around for a little while longer. Additionally, objects that are pinned into the cache are not counted when determining if the cache size exceeds the maximum.

Individual classes can be excluded from the data cache by setting the `can-cache` metadata extension to `false`.

The `DataCache` API provides a mechanism for pinning objects into memory by creating hard references to them. Caching algorithms are not permitted to flush objects that have been pinned from memory unless an explicit `remove()` call is made. To pin an object into memory, obtain a reference to the cache and invoke `pin()` on it:

Example 7.1. Pinning an object into the DataCache

```
PersistenceManagerFactoryImpl factory =
    (PersistenceManagerFactoryImpl) pm.getPersistenceManagerFactory ();
factory.getConfiguration ().getDataCache ().pin (JDOHelper.getObjectId (o));
```

A previously pinned object can later be unpinned by invoking `DataCache.unpin()`:

Example 7.2. Unpinning an object from the DataCache

```
PersistenceManagerFactoryImpl factory =
    (PersistenceManagerFactoryImpl) pm.getPersistenceManagerFactory ();
factory.getConfiguration ().getDataCache ().unpin (JDOHelper.getObjectId (o));
```

It is possible to evict data from the cache, but cache eviction does not automatically happen when the `evict()` method is invoked on a `PersistenceManager`. Instead, you must obtain a reference to the `DataCache` object from a `PersistenceManagerFactory` and explicitly evict the object id from the cache:

Example 7.3. Evicting an object from the DataCache

```
PersistenceManagerFactoryImpl factory =
    (PersistenceManagerFactoryImpl) pm.getPersistenceManagerFactory ();
factory.getConfiguration ().getDataCache ().remove (JDOHelper.getObjectId (o));
```

The JMS cache can be configured by setting the `com.solarmetric.kodo.DataCacheProperties` to contain the appropriate configuration properties. The JMS cache understands the following properties:

- Topic

Default: `topic/KodoCacheTopic`

Description: The topic that the cache should publish cache updates to and subscribe to for cache updates sent from other JVMs.

- TopicConnectionFactory

Default: `java:/ConnectionFactory`

Description: The JNDI name of a `javax.naming.TopicConnectionFactory` factory to use for finding topics.

To configure a `PersistenceManagerFactory` to use the JMS cache, your properties filename might look like the following:

Example 7.4. Configuring a `PersistenceManagerFactory` to use a JMS cache update mechanism

```
com.solarmetric.kodo.DataCacheClass=com.solarmetric.kodo.datacache.plugins.JMSCache
com.solarmetric.kodo.DataCacheProperties=Topic=topic/KodoCacheTopic CacheSize=5000
```

The Tangosol Coherence cache can be configured by setting the `com.solarmetric.kodo.DataCacheProperties` to contain the appropriate configuration properties. The Tangosol cache understands the following properties:

- TangosolCacheName

Default: `kodo`

Description: The name of the Tangosol Coherence cache to use.

- TangosolCacheType

Default: `distributed`

Description: The type of Tangosol Coherence cache to use. Valid values are either `distributed` or `replicated`.

To configure a `PersistenceManagerFactory` to use the Tangosol cache, your properties filename might look like the following:

Example 7.5. Configuring a `PersistenceManagerFactory` to use a Tangosol cache for distributed cache needs

```
com.solarmetric.kodo.DataCacheClass=com.solarmetric.kodo.datacache.plugins.TangosolCache
com.solarmetric.kodo.DataCacheProperties=TangosolCacheName=kodo TangosolCacheType=distributed
```

Note that as of this writing, it is not possible to use a Tangosol Coherence 1.2.2 distributed cache type with Apple's 1.3.1 JVM. Use their replicated cache instead.

The TCP and UDP caches have several options that are defined as host specifications containing a host name or ip address and an optional port separated by a colon. For example, the host specification `saturn.solarmetric.com:1234` represents an `InetAddress` retrieved by invoking `InetAddress.getByName ("saturn.solarmetric.com")` and a port of 1234.

The TCP cache can be configured by setting the `com.solarmetric.kodo.DataCacheProperties` to contain the appropriate configuration properties. The TCP cache understands the following properties:

- Port

Default: 5636

Description: The TCP port that the cache should listen on for data updates.

- Addresses

Default: none

Description: A semicolon-separated list of IP addresses to which invalidations should be sent. No default value.

To configure a `PersistenceManagerFactory` to use the TCP cache, your properties filename might look like the following:

Example 7.6. Configuring a `PersistenceManagerFactory` to use a TCP cache update mechanism

```
com.solarmetric.kodo.DataCacheClass=com.solarmetric.kodo.datacache.plugins.TCPCache
com.solarmetric.kodo.DataCacheProperties=Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13 CacheSize
```

The UDP cache can be configured by setting the `com.solarmetric.kodo.DataCacheProperties` to contain the appropriate configuration properties. The UDP cache understands the following properties:

- Port

Default: 5636

Description: The UDP port that the cache should listen on for data updates.

- PacketLength

Default: 1024

Description: The maximum packet length for which the cache should prepare a buffer.

- UseMulticast

Default: false

Description: A boolean selector for controlling how the cache communicates with other cache instances. If `true`, the cache will broadcast changes on the multicast host specification specified in the `multicast-group` pref, and will disregard the setting of `Port`. Otherwise, it will send a UDP packet to each IP address listed in the `addresses` list.

- `MulticastGroup`

Default: none

Description: The host specification of the multicast group to which packets should be sent.

- `Addresses`

Default: none

Description: A semicolon-separated list of IP addresses to which invalidations should be sent. No default value. Note that it is more efficient to use the multicast mechanism, as less network traffic is necessary.

To configure a `PersistenceManagerFactory` to use the UDP cache, your properties filename might look like the following:

Example 7.7. Configuring a `PersistenceManagerFactory` to use a UDP cache update mechanism

```
com.solarmetric.kodo.DataCacheClass=com.solarmetric.kodo.datacache.plugins.UDPCache
com.solarmetric.kodo.DataCacheProperties=Multicast=false Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13
```

7.3.3. Cache Extension

The provided data cache classes can be easily extended to add additional functionality. If you are adding new behavior, you should extend `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`, `com.solarmetric.kodo.runtime.datacache.plugins.JMSCache`, `com.solarmetric.kodo.runtime.datacache.plugins.TCPCache`, or `com.solarmetric.kodo.runtime.datacache.plugins.UDPCache`, as appropriate. If you want to implement a distributed cache that uses a method other than UDP for communications, extend the abstract class `com.solarmetric.kodo.runtime.datacache.plugins.DistributedCache`.

For specific examples of data cache extensions, please look at the updating JMS cache sample in the Kodo JDO distribution, or contact SolarMetric at jdosupport@solarmetric.com.

7.3.4. Important notes about the `DataCache`

- The default cache implementations *do not* automatically refresh objects in other `PersistenceManager` objects when the cache is updated or invalidated. This behavior would not be compliant with the specification. An example of how to extend the JMS cache to update all non-transactional `PersistenceManager` objects associated with a particular cache is available in the Kodo JDO distribution `samples/` directory.
- Invoking `PersistenceManager.refresh()` or `PersistenceManager.evict()` or related methods *does not* result in the corresponding data being dropped from the `DataCache`. The `DataCache` assumes that it is

up-to-date with respect to the data store, so it is effectively an in-memory extension of the data store. If you really want to force data out of the cache, you should use the `DataCache` APIs (see the `DataCache` JavaDoc for details), not the `PersistenceManager` cache control APIs.

- The `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache` does not communicate with other caches in the same JVM but associated with different `PersistenceManagerFactory` objects. However, if you make multiple calls to `JDOHelper.getPersistenceManagerFactory()` with the same `Properties` argument, then we ensure that all returned `PersistenceManagerFactory` objects are the same.
- Because of the nature of JMS, it is important that you invoke `PersistenceManagerFactoryImpl.close()` when finished with a `PersistenceManagerFactory` and all its `PersistenceManager` objects. If you do not do so, a daemon thread will stay up in the JVM, preventing the JVM from exiting.

7.3.5. Known issues and limitations

- When using data store (pessimistic) transactions in concert with the distributed caching implementations, it is possible to read stale data.

For example, if you have two JVMs (JVM A and JVM B) both communicating with each other, and JVM A obtains a data store lock on a particular object's underlying data, it is possible for JVM B to load the data from the cache without going to the data store, and therefore load data that should be locked. This will only happen if JVM B attempts to read data that is already in its cache during the period between when JVM A locked the data and JVM B received and processed the invalidation notification.

This problem is impossible to solve without putting together a two-phase commit system for cache notifications, which would add significant overhead to the caching implementation. As a result, we recommend that people use optimistic locking when using data caching. If you do not, then understand that some of your non-transactional data may not be consistent with the data store.

Note that when loading objects in a transaction, the appropriate data store transactions will be obtained. So, transactional code will maintain its integrity.

7.4. Query Extensions

JDOQL is designed to be back-end agnostic, meaning that the query language can be translated into any of a number of back-end languages. This is important for the JDO specification, because it permits true portability among fundamentally different data stores. However, when dealing with a relational data store, it is often necessary to use SQL-specific operations to harness the power and features of the relational data store. Kodo JDO Enterprise Edition allows developers to optimize and customize their datastore queries by adding new custom operations to JDOQL. In addition, Kodo provides some built-in extensions to simplify some common queries outside the JDOQL specification.

7.4.1. Using extensions in queries

To enable query extensions, set the `com.solarmetric.kodo.EnableQueryExtensions` property to `true`. This will enable query extensions and add the standard Kodo extensions listed below.

Extensions are used in filters with the given format:

- *ext:tagname(argument)*
- *lit:tagname(argument)*

Extensions thus begin with `ext` or `lit`, then a colon `:`, and then the extension tag name. Extensions also take a single constant argument.

Extensions can be either *Methods* or *Literals*. Literals can be thought of as self-contained SQL WHERE expressions which by themselves can evaluate to a SQL boolean value. Methods, on the other hand, act upon a given field in the query.

For example, to use Kodo's `stringContains` extension (which is a Method) one would create a filter that would look like the following code fragment that searches for cities with the substring "ford" (e.g. Hartford):

Example 7.8. Using Kodo's `StringContains` extension

```
Query q = pm.newQuery (City.class, true);
q.setIgnoreCache (true); // extensions work purely upon the datastore
q.setFilter ("name.ext:stringContains (\"ford\")");
Collection c = (Collection) q.execute ();
```

7.4.2. Included Kodo Extensions

Kodo includes some default extensions which are detailed below

- *lit:sqlEmbed*
Embeds argument right into the where clause as-is
- *ext:stringContains*
Searches for fields that contain a substring. Uses `String.indexOf()` when checking in-memory, and `colname LIKE %argument%` when compiling to SQL.
- *ext:caseInsStarts*
Requires DB support for `UPPER()`. Translates to `UPPER (column) LIKE UPPER (argument%)`
- *ext:caseInsEnds*
Requires DB support for `UPPER()`. Translates to `UPPER (column) LIKE UPPER (%argument)`
- *ext:caseInsContains*
Requires DB support for `UPPER()`. Translates to `UPPER (column) LIKE UPPER (%argument%)`

7.4.3. Writing custom extensions

Developers can write their own extensions by implementing one of two interfaces:

- `com.solarmetric.kodo.impl.jdbc.query.MethodListener`: A listener that will listen on its tag for filters calling *ext*.

The above `stringContains` implements this interface. The primary method to implement is `transform (String column, String argument)`. This method takes a given column and the argument passed to the filter and returns the SQL based upon those two parameters.

- `com.solarmetric.kodo.impl.jdbc.query.LiteralListener`: A listener that will listen on its tag for filters calling *lit*. Kodo's `sqlEmbed` extension implements this interface. The primary method to implement is `embed (String argument)` which return sql to embed into the WHERE clause.

In addition each listener must also provide Kodo with the tag that it is listening for with `public String getTag()`. If the listener is being loaded via properties (see below), it also should be instanceable by reflection (i.e. public empty constructor).

7.4.4. Configuring Extensions

There are primarily two means of registering user extensions with Kodo's querying system, each with benefits and tradeoffs depending on the application situation. The two can be mixed and matched depending on the needs of the application. However, Kodo extensions are loaded automatically by the system.

- *Registration by properties*

Listeners are registered by setting the `com.solarmetric.kodo.QueryFilterListeners` property to a comma separate list of listener classes. Listeners registered in this fashion must be able to be loaded via reflection. The extensions handled by these listeners apply for all datastore queries associated with a given `PersistenceManagerFactory`. This is a good way to install common listeners.

- *Runtime query by query basis*

Extensions can be registered with a given query via the `com.solarmetric.kodo.query.extensions.ExtensionHelper` helper class as shown in the following code fragment:

Example 7.9. Filter extensions usage example

```
public void addListener(Query q)
{
    ExtensionHelper.registerListener(q, new CaseListener());

    // lets have one that also listens on a separate tag
    ExtensionsHelper.registerListener(q, new CaseListener(CaseListener.UPPER));
}

public static class CaseListener implements MethodListener
{
    public static final int LOWER=0;
    public static final int UPPER=1;

    private int type = LOWER;

    public CaseListener () {}
    public CaseListener (int type)
    {
        this.type = type;
    }

    public String getTag ()
    {
        switch (type)
        {
```

```

        case UPPER:
            return "likeUpper";
        case LOWER:
        default:
            return "likeLower";
    }
}

public String transform (String column, String argument)
{
    switch (type)
    {
        case UPPER:
            return column + " LIKE UPPER (" + argument + ")";
        case LOWER:
        default:
            return column + " LIKE LOWER (" + argument + ")";
    }
}
}

```

When registering listeners on a query by query basis, listeners do not transfer from query to query. However, it is possible to do more customized construction of Listeners. For example, a listener could be constructed with complicated SQL based on application data which only gets hints from the argument.

7.5. Fetch Groups

The JDO specification defines a concept of a default fetch group, but it does not touch upon additional, non-default fetch groups.

7.5.1. Normal Default Fetch Group Behavior

First, let's talk about how Kodo JDO behaves when loading data with just the regular JDO default fetch group information. Imagine the following class and metadata definitions:

```

public class FetchGroupExample
{
    private int                a;
    private String             b;
    private BigInteger          c;
    private Date               d;
    private String             e;
    private String             f;
    private FetchGroupExample g;
}

```

```

<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="FetchGroupExample">
      <field name="a"/>
      <field name="b"/>
      <field name="c"/>
      <field name="d"/>
      <field name="e"/>
      <field name="f"/>
      <field name="g"/>
    </class>
  </package>
</jdo>

```

```

        </class>
    </package>
</jdo>

```

In this example, the default fetch group behavior is left undefined for all fields. So, the default values defined in the JDO specification will be used: all fields except `g` will be in the default fetch group. `g` will be left out of the default fetch group because it is a reference to another persistence-capable object.

Kodo JDO will load all fields in the object in the initial select statement, including the primary key of `g`. This primary key will be loaded because the related object may already be in the `PersistenceManager`'s cache, so we may be able to set up this relation up-front, and since we're already going to the database for all the other fields, we might as well check the primary key. In general, this behavior is ideal, since the cost of executing a select statement including the extra fields for one-one relations from the database is minimal compared to the cost of going back to the database for this information when it's needed.

However, in some situations, it is undesirable to load certain parts of an object up-front. Sometimes, a table in the database will be comprised of many columns, so selecting the extra data -- especially if the returned result set is expected to be large -- can impose a significant overhead. Imagine loading all fields in all `Employee` objects associated with a large company when generating a report listing all employees. All we really needed might have been employee number and name, so loading the entire object up-front could incur a quite significant amount of unneeded data to be transferred.

To improve upon this situation, the extra fields could be defined to not be in the object's default fetch group. By doing this, the developer is providing a hint to the JDO implementation that the identified data should be lazily loaded, rather than materialized at initialization time. (In the above example, had we explicitly excluded field `g` from the default fetch group, Kodo would not have loaded the primary key values for this field.)

Kodo JDO's handling of fields implicitly excluded from the default fetch group is a bit more complex when dealing with multiple-table class inheritance hierarchies. As mentioned above, Kodo loads the primary keys for implicitly excluded fields when selecting data from the database. This extra data loading is not performed if the column holding the data is in a table that would not otherwise be selected. That is, we do not add an extra join in order to load this data.

Chapter 8. Additional Features

Kodo JDO Enterprise Edition can be extended and customized to match particular business needs. Some commonly requested customizations are available as fully supported add-on features. This chapter documents these features.

8.1. Custom data processing

Sometimes, data must be loaded from non-standard input feeds, such as from XML input or from `ResultSets` obtained from custom stored procedures. Kodo JDO's `CustomResultObjectProvider` interface can be used to load arbitrary data into `PersistenceCapable` objects associated with a particular `PersistenceManager`.

Documentation of these interfaces, including some code fragments, can be found in the JavaDoc for the `com.solarmetric.kodo.runtime.objectprovider` package, and the `CustomResultSetResultObjectProvider` and `ColumnAliasResultObjectProvider`. Additionally, an example of how to use the `CustomResultObjectProvider` interface to load externally obtained SQL data is available in the Kodo JDO distribution.

The `ColumnAliasResultObjectProvider` implementation is useful when loading data via a custom `SELECT` statement from tables that Kodo JDO is already configured to map to. In this situation, the Kodo JDO metadata contains enough information for Kodo JDO to do all the data loading of the result set without any application-specific code.

When more application-specific result sets must be processed, the `CustomResultSetResultObjectProvider` abstract class should be extended. The subclass must provide an implementation of the `getFieldValues` method, which creates a `Map` of field names to field values, which will then be loaded into a `StateManagerImpl` by the Kodo JDO framework.

8.2. Custom data requests

The custom data processing feature described above is useful when loading data from externally obtained data feeds, but often, this is only half of the story. Once that data is initially loaded, requests to reload all or part of a particular object may be made by the Kodo JDO runtime framework. Additionally, requests to insert, update or delete data may be issued.

Kodo JDO often makes requests to the JDBC subsystem to find information about a particular object. For example, when a user invokes `PersistenceManager.getObjectById`, Kodo JDO will make a request to the JDBC subsystem to load that particular object. Similarly, Kodo JDO will occasionally make a request to see if a particular object exists in the data store, or if the optimistic lock version information indicates an optimistic lock violation. To intercept these calls and provide custom logic to load the data, extend `com.solarmetric.kodo.impl.jdbc.ormapping.ClassMapping` and override the appropriate methods. Following is a brief discussion of some commonly overridden APIs in `ClassMapping`.

- `insert`: Responsible for storing a new persistence-capable instance in the data store.
- `update`: Responsible for updating the data store with the dirty fields in the given `StateManagerImpl`.
- `delete`: Responsible for removing the given `StateManagerImpl` from the database.
- `newExtent`: Responsible for creating a new `Extent` capable of iterating through all objects of the type managed by the `ClassMapping`.
- `loadByPK`: Responsible for selecting and loading the requested data into the provided `StateManagerImpl`.

Note that the default relation mappings (one-one, one-many, many-many, and n-many) do not necessarily invoke

`loadByPk`. Instead, they directly load data from result sets. So, if data must be loaded purely through non-standard means, special care must be taken when loading these types of relations. (One-one relations may be loaded via calls to `loadByPk` if the foreign key information is loaded at initialization time.)

- `checkVersion`: Determines if another thread has modified the data represented by the given `StateManagerImpl`. This is used when determining if an optimistic lock exception should be thrown.
- `exists`: Invoked to check if the given `StateManagerImpl` exists in the data store.

Alternate `ClassMapping` implementations can be specified on a system-wide basis using the `DefaultClassMappingClass` property, or on a per-class basis with the custom-mapping metadata extension.

Chapter 9. Third Party Integration Features

9.1. Overview of Third Party Integration features in Kodo

Kodo provides a number of mechanisms for integrating with third-party tools. The following chapter will illustrate these integration features.

9.2. Apache Ant

Ant is a very popular tool for building java projects. It is similar to the make command, but is java-centric and has more modern features. Ant is open-source, and can be downloaded from Apache's Ant web page at <http://jakarta.apache.org/ant/>. Ant has become the de-facto standard build tool for java, and many commercial integrated development environments provide some support for using ant build files. The remainder of this section assumes familiarity with writing Ant `build.xml` files.

Kodo provides three pre-built Ant task definitions for use in `build.xml` files:

- XDoclet: Allows you to auto-generate `.jdo` metadata files from java source code that includes certain well-formed comments.
- JDOEnhancer: Allows the JDOEnhancer to be invoked directly from within ant.
- SchemaTool: Allows the SchemaTool to be invoked directly from within ant.

The source code for all the ant tasks is provided with the distribution under the `source/` directory. This allows developers to customize various aspects of the ant tasks in order to better integrate into their development environment.

9.2.1. Common Ant Configuration Options

Both the JDOEnhancer task and the SchemaTool task can take a nested `<config>` element, which defines the configuration environment in which the specified task will run. The attributes for the `<config>` tag are defined by the JDBCConfiguration bean methods. Note that excluding the `<config>` element will cause the Ant task to use the default system configuration mechanism, such as the configuration defined in the `kodo.properties` file.

Following is an example of how the nested `<config>` tag can be used in a `build.xml` file:

Example 9.1. Using the `<config>` tag in an ant `build.xml` file

```
<schematool action="refresh" ignoreErrors="true">
  <fileset dir="${basedir}">
    <include name="**/*.jdo" />
  </fileset>
  <config connectionUsername="scott" connectionPassword="tiger"
    licenseKey="1234-5678-90ab-cdef"
    connectionUrl="jdbc:oracle:thin:@saturn:1521:solarsid"
    connectionDriverName="oracle.jdbc.driver.OracleDriver" />
</schematool>
```

The JDOEnhancer task and the SchemaTool task can also take a nested `<classpath>` element, which can be used if the default classpath is not desired. The `<classpath>` argument behaves the same as it does for ant's standard

`javac`> element. It is sometimes the case that projects are compiled to a separate directory than the source tree. If the target path for compiled classes is not included in the project's classpath, then a `<classpath>` element that includes the target class directory needs to be included in enhancer and schematool tags so that the tools can locate the classes.

Following is an example of using a `<classpath>` tag:

Example 9.2. Using the `<classpath>` tag in an ant build.xml file

```
<jdoc>
  <fileset dir="${basedir}/source">
    <include name="**/*.jdo" />
  </fileset>
  <classpath>
    <pathelement location="${basedir}/classes"/>
    <pathelement location="${basedir}/source"/>
    <pathelement path="${java.class.path}"/>
  </classpath>
</jdoc>
```

9.2.2. JDOEnhancer Ant Task

The JDOEnhancer Ant task allows you to invoke the JDOEnhancer directly from within the Ant build process. It takes a nested `<fileset>` tag to specify the `.jdo` that should be processed. For more details on the JDOEnhancer, see the enhancer section.

Following is an example of using the JDOEnhancer task in a `build.xml` file:

Example 9.3. Invoking the JDOEnhancer from an Ant build.xml file

```
<target name="enhanceAll">
  <!-- Define the jdoc task definition. This can
       be done at the top of the build.xml file,
       so it will be available for all targets. -->
  <taskdef name="jdoc"
    classname="com.solarmetric.modules.integration.ant.JDOEnhancerTask"/>

  <!-- Invoke the JDOEnhancer on all .jdo files
       below the current directory. -->
  <jdoc>
    <fileset dir=".">
      <include name="**/*.jdo" />
    </fileset>
  </jdoc>
</target>
```

9.2.3. SchemaTool Ant Task

The SchemaTool Ant Task allows you to directly invoke the SchemaTool from within the Ant build process. It is useful for refreshing a development database after refactoring a `.jdo` metadata file without needing to remember to invoke `schematool` manually each time.

The task accepts the following parameters:

- `action`: can be one of "add", "refresh", or "drop". The meaning of these attributes is identical to the value of the "-action" flag for the `schematool` command.
- `ignoreErrors`: a value of "false" will cause the build process to abort if there are any SQL errors when communicating with the data store.
- `outputFile`: If specified, output generated SQL to a file instead of sending it to the database.

Following is an example of a `build.xml` target that invokes the `SchemaTool`:

Example 9.4. Invoking the `SchemaTool` from an Ant `build.xml` file

```
<target name="refreshDataStore">
  <!-- Define the schematool task definition. This can
       be done at the top of the build.xml file,
       so it will be available for all targets. -->
  <taskdef name="schematool"
    classname="com.solarmetric.modules.integration.ant.SchemaToolTask"/>

  <!-- Locate all the .jdo files below this, and
       refresh the data store's schema to be in synch
       with the current state of the metadata. -->
  <schematool action="refresh" ignoreErrors="true">
    <fileset dir=".">
      <include name="**/*.jdo" />
    </fileset>
  </schematool>
</target>
```

9.3. XDoclet

XDoclet is an open-source project hosted at <http://xdoclet.sourceforge.net>. It is an extension of Sun's javadoc tool that allows you to embed well-formed tags in java source code and output metadata of a particular type. In the case of the JDODoclet, you can generate a `filename.jdo` file based on various tags starting with `@jdo:`. Any class that has the `@jdo:persist` class-level tag will be processed. This allows for all refactoring to be done in just one place (the java source code file), without the developer needing to manually ensure that the `.jdo` metadata file is always synchronized with the state of the java source code.

In order to utilize the XDoclet task, you will need to download the XDoclet libraries separately from <http://xdoclet.sourceforge.net> and add them to your CLASSPATH. The JDODoc task can then be executed from the `build.xml` as follows:

Example 9.5. Invoking the JDO Metadata generator from an Ant `build.xml` file

```
<target name="generateJdoMetadata">
  <taskdef name="jdodoclet"
    classname="com.solarmetric.modules.integration.ant.KodoDocletTask"/>

  <jdodoclet sourcepath="${basedir}" destdir="${classes}">
    <fileset dir="${basedir}">
      <include name="**/*.java"/>
    </fileset>

    <!-- perform the actual transformation here -->
    <jdotags/>
  </jdodoclet>
</target>
```

```
</jdodoclet>
</target>
```

An example of a commented file is as follows:

Example 9.6. Source code comments for automatic JDO Metadata generation

```
package samples.xdoclet;

import java.util.*;

/**
 * This is the TestDoclet example class. It is used to
 * demonstrate automatic generation of the
 * TestDoclet.jdo file based on the jdo: tags
 * in the source code.
 *
 * Note that the double-$ for inner class names is required
 * in order to escape ant variable handling.
 *
 * The jdo:persist tag is required for the XDoclet to know that we
 * should generate persistence information for this class.
 *
 * @jdo:persist
 * @jdo:identity-type      application
 * @jdo:objectid-class     TestDoclet$$Id
 * @jdo:requires-extent    false
 * @jdo:extension          vendor-name="kodo" key="table"
 *                        value="DOCLET_TEST_TABLE"
 * @jdo:extension          vendor-name="kodo" key="pk-column"
 *                        value="TEST_PK_COLUMN"
 * @jdo:extension          vendor-name="kodo" key="lock-column"
 *                        value="TEST_LOCK_COLUMN"
 * @jdo:extension          vendor-name="kodo" key="class-column"
 *                        value="TEST_CLASS_COLUMN"
 *
 * @author                 Marc Prud'hommeaux
 */
public class TestDoclet
{
    /**
     * @jdo:primary-key      true
     * @jdo:extension        vendor-name="kodo" key="column-length"
     *                        value="10"
     */
    private String pk1;

    /**
     * @jdo:primary-key      true
     * @jdo:extension        vendor-name="kodo" key="column-length"
     *                        value="20"
     */
    private String pk2;

    /**
     * @jdo:persistence-modifier  transactional
     * @jdo:null-value            exception
     * @jdo:default-fetch-group    true
     * @jdo:embedded              true
     * @jdo:extension            vendor-name="kodo" key="data-column"
     *                        value="NAME_DATA"
     */
    private String name;

    /**
```

```

    * @jdo:extension      vendor-name="kodo" key="data-column"
    *                    value="MEMO_COLUMN"
    * @jdo:extension      vendor-name="kodo" key="column-length"
    *                    value="-1"
    */
private String memo;

/**
 * @jdo:extension      vendor-name="kodo" key="data-column"
 *                    value="AGE_DATA"
 */
private int age;

/**
 * @jdo:collection      element-type="TestDocletChild"
 *                    embedded-element="false"
 * @jdo:extension      vendor-name="kodo" key="table"
 *                    value="TEST_XREF_DOCLET_TABLE"
 * @jdo:extension      vendor-name="kodo" key="ref-column"
 *                    value="TEST_CHILDREN_REF_COLUMN"
 * @jdo:extension      vendor-name="kodo" key="order-column"
 *                    value="TEST_CHILDREN_ORDER_COLUMN"
 * @jdo:extension      vendor-name="kodo" key="inverse"
 *                    value="childInverse"
 */
private Collection children = new ArrayList ();

/**
 * @jdo:map              key-type="String" embedded-key="false"
 *                    value-type="Integer" embedded-key="false"
 * @jdo:extension      vendor-name="kodo" key="key-column"
 *                    value="TEST_MAP_KEY"
 */
private Map testMap = new HashMap ();

/**
 * Application identity class.
 */
public static class Id
{
    public String pk1;
    public String pk2;

    public boolean equals (Object other)
    {
        return other.getClass () == this.getClass () &&
            ((Id)other).pk1.equals (pk1) &&
            ((Id)other).pk2.equals (pk2);
    }

    public int hashCode ()
    {
        return pk1.hashCode () + pk2.hashCode ();
    }
}
}

```

The resulting TestDoclet.jdo metadata file will then look like:

```

<?xml version="1.0" encoding="UTF-8"?>
<!--
  This file is auto-generated by the
  com.solarmetric.modules.integration.ant.JDODoclet ant task.
  Bear in mind that any changes to this file will be overwritten
  if the task is re-run.
-->
<jdo>
  <package name="samples.xdoclet">
    <class name="TestDoclet" identity-type="application"

```

```

objectid-class="TestDoclet$Id" requires-extent="false" >
  <extension vendor-name="kodo" key="table"
    value="DOCLET_TEST_TABLE" />
  <extension vendor-name="kodo" key="pk-column"
    value="TEST_PK_COLUMN" />
  <extension vendor-name="kodo" key="lock-column"
    value="TEST_LOCK_COLUMN" />
  <extension vendor-name="kodo" key="class-column"
    value="TEST_CLASS_COLUMN" />
  <field name="age">
    <extension vendor-name="kodo" key="data-column"
      value="AGE_DATA" />
  </field>
  <field name="children">
    <collection element-type="TestDocletChild" embedded-element="false" />
    <extension vendor-name="kodo" key="table"
      value="TEST_XREF_DOCLET_TABLE" />
    <extension vendor-name="kodo" key="ref-column"
      value="TEST_CHILDREN_REF_COLUMN" />
    <extension vendor-name="kodo" key="order-column"
      value="TEST_CHILDREN_ORDER_COLUMN" />
    <extension vendor-name="kodo" key="inverse"
      value="childInverse" />
  </field>
  <field name="memo">
    <extension vendor-name="kodo" key="data-column"
      value="MEMO_COLUMN" />
    <extension vendor-name="kodo" key="column-length"
      value="-1" />
  </field>
  <field persistence-modifier="transactional" null-value="exception"
    default-fetch-group="true" embedded="true" name="name">
    <extension vendor-name="kodo" key="data-column"
      value="NAME_DATA" />
  </field>
  <field primary-key="true" name="pk1">
    <extension vendor-name="kodo" key="column-length"
      value="10" />
  </field>
  <field primary-key="true" name="pk2">
    <extension vendor-name="kodo" key="column-length"
      value="20" />
  </field>
  <field name="testMap">
    <map key-type="String" embedded-key="false" value-type="Integer" />
    <extension vendor-name="kodo" key="key-column"
      value="TEST_MAP_KEY" />
  </field>
</class>
</package>
</jdo>

```

9.4. Borland JBuilder

Kodo JDO provides integration into JBuilder 7 and higher in the form of a JBuilder OpenTool. The integration features allow the JBuilder user to configure the Kodo runtime, edit .jdo metadata files (both as raw XML and via a specialized editor), automatically run the JDO Enhancer as part of the build process, and perform various schema manipulation tasks.

9.4.1. Installing Kodo into JBuilder

To install Kodo support in JBuilder, just copy all the .jar files from the lib/ directory of your Kodo installation to the lib/ext/ directory of JBuilder, and copy the lib/KodoJDO.library to JBuilder's lib/ directory. For example, if Kodo is installed in C:\development\kodo\ and JBuilder is installed in C:\JBuilder7\, then

you would copy all the `.jar` files from `C:\development\kodo\lib\` to `C:\JBuilder7\lib\ext\`, and then copy the `C:\development\kodo\lib\KodoJDO.library` file to `C:\JBuilder7\lib\`.

To validate the installation, you should start (or restart) JBuilder. You should see the Kodo logo in the build toolbar, which is used to configure the Kodo installation.

Note

If you use the Windows Installer program to install Kodo, and you elected to perform the "Install Kodo JBuilder extensions", then you do not need to perform the manually file copying or any other additional steps.

Warning

The Kodo JBuilder OpenTool only works in JBuilder 7 and 8. It will not work in releases of JBuilder prior to version 7.

9.4.2. Kodo Configuration from JBuilder

The Kodo configuration panel provides various options for configuring runtime usage of Kodo. Configuration options are saved in JBuilder's user `.properties`, and will be automatically written out to a file named `kodo.properties` at the root of the project directory for runtime configuration operations. The current project's `kodo.properties` file can either be edited manually, or values can be modified in the "Project Configuration" tab of the Kodo Configuration dialog.

9.4.3. Creating and building JDO projects in JBuilder

When right-clicking on a `.java` in JBuilder, you will see an option to "Create Kodo JDO Metadata". This will create a default `.jdo` file for the selected `.java` file. The `.jdo` file that is created will then be edit-able in the JBuilder interface, either manually as an XML text file, or via the integrated metadata editor. For more details on JDO metadata, please see the Metadata section.

Note

Creating and editing package `.jdo` metadata files for multiple classes is currently not supported in the JBuilder interface.

The JDO enhancer will be automatically run on any `.class` file that has an associated `.jdo` metadata file during the JBuilder build process. Furthermore, the `.jdo` file will be copied over to the output directory, so that it will be available at runtime.

9.4.4. Editing JDO Metadata from JBuilder

The `.jdo` metadata files can either be edited in JBuilder's native XML editor, or they can be modified using a dialog by selecting "Properties" from the context menu of the `.jdo` file in the JBuilder browser. The dialog contains entries for all of the standard JDO attributes, as well as Kodo-specific vendor extensions. See the section on JDO Metadata for more details about the various properties and their meanings.

9.4.5. Running the SchemaTool from JBuilder

The Schema Tool can be run from within JBuilder on one or more `.jdo` metadata files by selecting them in the JBuilder browser pane and selecting "Kodo JDO Database Schema Tool" from the context menu. The Schema Tool GUI allows users to perform all the operations of the command-line schema tool. See the Schema Manipulation for

more details on the Schema Tool.

9.4.6. JBuilder Project Sample

An example JBuilder Swing project is available in the Kodo JDO installation, in the `samples/swing/petshop` directory. To run the sample, do the following:

- Double-click the `PetShop.jpx` file in the `samples/swing/petshop` directory of your Kodo JDO installation. This will load the PetShop sample in JBuilder.
- Build the project by selecting `Make Project "PetShop.jpx"` from the `Project` menu. This will also run the JDO enhancer on `Animal`.
- Expand the `<Project Source>` item in the top left pane of the JBuilder display. This will expose the classes in the sample and the `Animal.jdo` metadata file.
- Right-click on the `Animal.jdo` file in the top left pane, and select `Kodo JDO Database Schema Tool` from the menu. This will load the schema tool dialog.
- Click on the `Execute` button at the bottom of the dialog, and then click `OK`. This will initialize the database with the appropriate table for the `Animal` class.
- Right-click on the `PetShop.java` file in the top left pane, and select `Run using defaults` from the menu. This will run the Pet Shop example.

The Pet Shop example allows you to create and delete pets in a database. The pets have a string type -- dog, cat, giraffe, etc. -- and a price.

The Pet Shop example code demonstrates how to put together a simple Swing example, and also how to use a Kodo-specific feature to extend the `PersistenceManager` class to enable Swing updates to happen at the optimal time.

9.5. Sun ONE Studio / NetBeans IDE

Kodo JDO can integrate with both Sun ONE Studio and NetBeans IDE as an OpenIDE module. The module requires versions 4.0 and 3.4 or higher of Sun ONE Studio and NetBeans IDE respectively. The module provides support for integrating jdo metadata files into the IDE, easy access to the SchemaTool and enhancer, links into the build process to support enhancement, and wizards to help in the creation of JDO specific files.

9.5.1. Before Installing Kodo into the IDE

This document will refer to your IDE's home directory, for example, `C:\Program Files\s1studio` or `/usr/local/NetBeansIDE_3.3`. If you are installing on a multi-user installation and want to install it for only a particular user, this home IDE directory can also be the user's directory chosen during the initial execution of the IDE, for example `C:\ide-userdir`. On Linux and Unix, this folder usually automatically set to `~/ffjuser40` and `~/netbeans/3.4` for SunONE Studio and NetBeans IDE respectively.

Previous Kodo jars should be removed from the IDE's classpath. If Kodo jars had been mounted into the IDE filesystem, unmount those jars in the IDE before continuing. Then remove these jars from the `lib` and `lib/ext` directories of both the user and system IDE home directories before continuing if they are present..

9.5.2. Installing Kodo into the IDE

First copy the following jars into the `lib/ext` directory of your IDE's home directory (note the version numbers

may be different in your Kodo installation). Do *not* place the Kodo jars in this directory as we will install this later in the process. Unix and Linux users: do not symlink these jars as your permissions may be incorrect.

- `jakarta-commons-logging-1.0.2.jar`
- `xml-apis.jar`
- `jdbc2_0-stdext.jar`
- `jdo1_0.jar`
- `serp.jar`

In addition, one should include the jar for the JDBC driver used to connect to the database used for development. Place `kodo-jdo.jar` in your home IDE directory under `modules`.

Upon starting up the IDE, open the IDE configuration under the menu at `Tools -> Options`. If you have installed Kodo correctly, you should see Kodo listed as a module under `Options -> IDE Configuration -> System -> Modules`. If its not listed, right click on `Modules -> Add -> Module...` and browse to `kodo-jdo.jar` to manually install Kodo as a module.

To begin using the JDO API, open the Filesystems browser. Right click on filesystems, and select `Mount -> Archive` and browse to and select `jdo1_0.jar` which you installed. To explicitly use Kodo API extensions, mount `kodo-jdo.jar` in a similar fashion.

NetBeans users who experience a variety of `ClassNotFoundException`s for ant Kodo or other related jar-contained class may have to mount both `kodo-jdo.jar` as well as those installed at `lib/ext`. In addition, an XML SAX Parser should be mounted (e.g. `xerces.jar` in `lib/ext` of the IDE base directory) and configured (if not using Xerces).

9.5.3. Configuring the Kodo Module

Configuring Kodo is integrated into the IDE. In the Options dialog, Kodo's options are listed under `Options -> JDO -> Kodo Settings`. You must enter your license key, and configure the connection info to point to your development database.

You can optionally provide under Configuration Wizard Defaults defaults for new `kodo.properties` files. If you don't see the option, we have seen that the IDE often needs to be restarted after installation of new Modules for all options to be visible.

9.5.4. Kodo Template Wizards

The Kodo Module provides a pair of templates to help you get started and use Kodo and JDO. These templates are activated by right clicking on a folder in your project or filesystem, and selecting `New -> Kodo`.

The Kodo Properties wizard will assist in the run-time configuration of Kodo, both for development-time and deployment-time. The wizard will create new `.properties` files to use through `Properties.load()`. Options are conveniently separated into logical groupings. You can optionally test this new configuration by pressing the test button on the lower right corner

The JDO Metadata wizard provides a set of dialogs to walk the developer through metadata generation. In addition, to JDO standard options, most Kodo extensions are seamlessly integrated into the wizard. Simply select the type of metadata file to generate, add the classes you want described by the metadata, and then configure each class. The resulting `jdo` file can now be used both as a proper `.jdo` file and in your project as outlined below.

9.5.5. JDO DataObject

Kodo provides integrated support for JDO files as OpenIDE DataObjects. You can treat them like any other file. Right clicking on a JDO file marked by the Kodo "K" will present a variety of options. To edit the file in XML view, select `Edit`. To open and edit the file in a JDO Metadata editor, select `Open`.

In addition, the module integrates support for enhancing and running SchemaTool over classes in your project and in your metadata file. To accomplish either task, right click on any JDO Metadata file node, or Java class node in the explorer. You can select one or more of either node as long as no non-applicable nodes are selected. Select `Tools` -> `Kodo` to see the available tools.

SchemaTool will provide the additional option of modifying the database. Selecting `No` will simply walk through SchemaTool's actions without actually changing, creating, or dropping any database columns or tables.

9.5.6. Kodo Integration into the Build Process

By installing the Kodo Module, Kodo will integrate into the project build process. By adding your JDO files to the project, Kodo will make sure that your dependent classes are compiled before enhancement occurs.

Kodo also integrates via Ant. See the Ant integration section for more details.

9.5.7. SunONE / NetBeans Sample

To build and run the sample, first following installation instructions above. Mount the `samples/swing/petshop` directory for your Kodo JDO installation onto the `Filesystem` explorer pane (right click on `FileSystems`, select `Add Existing`, and browse and select the directory). In a similar manner, mount `kodo-jdo.jar` from the `modules` directory of the IDE home directory. To run the sample, do the following:

- Build the sample by selecting the `petshop` directory node (a root node), and selecting from the menu `Build` -> `Build All` This will also run the JDO enhancer on `Animal`.
- Expand the `petshop` directory node in the explorer. This will expose the classes in the sample and the `Animal.jdo` metadata file.
- Right-click on the `Animal.jdo` file in the top left pane, and select `Tools` --> `Kodo` --> `SchemaTool` - `Refresh` from the menu. Before selecting yes, ensure that your Kodo module configuration matches the `kodo.properties` in the current directory. By selecting to modify the database, this action will create the tables necessary for the sample. By selecting `No`, the SchemaTool will go through the motions *without* altering the true database schema.
- Right-click on the `PetShop` node in the explorer and select `Execute` from the menu. This will run the Pet Shop example.

The Pet Shop example allows you to create and delete pets in a database. The pets have a string type -- dog, cat, giraffe, etc. -- and a price.

The Pet Shop example code demonstrates how to put together a simple Swing example, and also how to use a Kodo-specific feature to extend the `PersistenceManager` class to enable Swing updates to happen at the optimal time.

9.6. Eclipse / WebSphere Studio Integration

The versions that the plugin has been tested are 1.0 and 2.0 of the Eclipse IDE, and version 4 and 5 of WebSphere

Studio Application Developer. While the plugin may work on other IDEs based upon Eclipse technology, they have not been tested and may not be fully supported.

Kodo JDO integrates as a plugin to IDEs based upon the open source Eclipse technology. The plugin provides quick access to many of Kodo's features, including metadata enhancement, SchemaTool, and building projects.

9.6.1. Installing the Kodo Eclipse Plugin

First copy the directory `com.solarmetric.kodo_1.0.0` directory from the eclipse directory in your Kodo installation to the plugins directory of the IDE base directory. For example, if you installed Kodo at `C:\Kodo` and Eclipse at `C:\Eclipse`, one would take `C:\Kodo\eclipse\com.solarmetric.kodo_1.0.0` directory and place it such that the new directory structure would be: `C:\Eclipse\plugins\com.solarmetric.kodo_1.0.0`.

Copy all the jars from the lib directory of your Kodo installation into this new directory. In addition, copy the JDBC driver of your choice into this directory as well. Keep the driver filename in mind as you will need it later.

In the Kodo plugin directory, modify the marked portion of `plugin.xml` to point to your JDBC driver .jar file.

Start the IDE. You should see a Kodo menu item, and Kodo listed as an available view. If not, you may need to configure your perspective to include those items. To manually configure your perspective for WSAD 4, perform the following steps. These steps are likely to work for other versions of Eclipse, but have been written with WSAD 4 in mind.

1. Click on Perspective->Customize... to open the "Customize Perspective" dialog box.
2. Open up the "Views" checkbox.
3. Select "Kodo" from the list of available views if it's unselected. This will add the Kodo debug view to the views available to your perspective.
4. Next, collapse the "Views" checkbox and open up the "Other" checkbox.
5. Select "Kodo Actions" if it's unselected. This will add the Kodo menu to the menu bar.
6. Hit OK to close out of the customization dialog box. You should now see the Kodo menu item, and a Kodo view in the Perspective->Views menu.

9.6.2. Configuring the Plugin

First, configure Kodo's development time options by editing your license key and database options. To do this, go into Window->Preferences and select Kodo Preferences. Edit to match your configuration and select apply, then ok. You should now be able to use the enhancer and schematool.

To have JDO and Kodo available to your program, you should add all the Kodo jars either from the plugin directory or your Kodo installation directory under your project properties.

9.6.3. Using Kodo in Eclipse IDEs

By selecting a file in a project, you can add and remove Kodo's enhancer to the project's build process. This builder will sequentially enhance any .jdo files *only on full project builds and rebuilds*.

You can also manually enhance your .jdo file by selecting it in the explorer pane and selecting either the Enhance icon in the toolbar or selecting Kodo->Enhance.

To run the SchemaTool, similarly select a .jdo file and select one of the SchemaTool icons or the corresponding menu item. A dialog will ask if you want to alter the schema. Selecting no will simulate altering the schema while printing the SQL to the Kodo view without actually executing any of it.

9.6.4. Eclipse Sample

To build and run the sample, first install the plugin following the instructions above. Create a new Java project or use an existing one. Right click on the project, select properties. In the properties window, select in the left navigation, Java Build Path. In the tabs that opens up, select the library tab. Press the Add External Jars... button and browse to your plugin directory or the lib directory in your Kodo installation. Select all the jars (in some operating systems, you may have to add them one at a time). and select ok.

Right click again on your project, but this time select Import (Some IDEs have this option only on the main File menu). Select File System on the screen that allows you to specify what sort of resources you are adding. Browse to the samples/swing/petshop directory. Note that you must select petshop, not examples or the Kodo installation directory inside the file browser. Select all the files in the directory, making sure to include .jdo and .properties files if they are filtered. Select OK and you should see Animal.java among other files inside a folder called default.package underneath your project.

First, right click on a source file such as Animal.java. Under the Kodo menu, select Add Enhancer To Build to add the enhancement process to the build. Now select Project->Rebuild All. This will compile the classes and enhance the metadata files in your project. Now run SchemaTool on Animal.jdo by selecting the file and either clicking on Kodo->SchemaTool->Add or selecting the icon with the matching tooltip. Note that you can manually enhance Animal.jdo in a similar manner.

Now alter kodo.properties to match your IDE configuration plus any other options you desire. Select Run->Run, and select Java Application. Press the New button and configure the IDE to run the PetShop class. Select Run and you should soon be seeing the PetShop Swing application run.

The Pet Shop example allows you to create and delete pets in a database. The pets have a string type -- dog, cat, giraffe, etc. -- and a price.

The Pet Shop example code demonstrates how to put together a simple Swing example, and also how to use a Kodo-specific feature to extend the PersistenceManager class to enable Swing updates to happen at the optimal time.

Chapter 10. Enterprise Integration

10.1. Overview of Enterprise Integration features in Kodo

Kodo can be integrated with your J2EE applications in a number of ways. The simplest is just to include the Kodo libraries in a J2EE EAR file and call Kodo directly from there. This will be problematic, however, if you want Kodo to share resources with other beans or if you have multiple beans deployed that need to use the same Kodo resource.

The second way to integrate Kodo is to configure and bind an instance of `com.solarmetric.kodo.impl.jdbc.ee.EEPersistenceManagerFactory` into JNDI. Although the mechanism by which you do this is up to you, the most common way is to create a startup class according to your application server's documentation that will create an instance of `EEPersistenceManagerFactory` and then bind it in JNDI. For example, in WebLogic you could write a startup class as follows::

Example 10.1. Binding a PersistenceManagerFactory into JNDI via a WebLogic startup class

```
package samples.weblogic.kodo;

import javax.naming.*;
import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import java.util.Hashtable;
import com.solarmetric.kodo.impl.jdbc.ee.EEPersistenceManagerFactory;

/**
 * This startup class created and binds an instance of a
 * Kodo EEPersistenceManagerFactory into JNDI.
 */
public class StartKodo
    implements T3StartupDef
{
    private static String myJNDIName = "my.jndi.name";
    private T3ServicesDef services;

    public void setServices (T3ServicesDef services)
    {
        this.services = services;
    }

    public String startup (String name, Hashtable args)
        throws Exception
    {
        String jndi = (String)args.get ("jndiname");
        if (jndi == null || jndi.length () == 0)
            jndi = myJNDIName;

        EEPersistenceManagerFactory factory =
            new EEPersistenceManagerFactory ();

        // you could perform additional configuration of the
        // EEPersistenceManagerFactory here. Otherwise, the values
        // from the Kodo properties or system.prefs will be used.

        InitialContext ic = new InitialContext ();
        ic.bind (jndi, factory);

        // return a message for logging
        return "Bound EEPersistenceManagerFactory to " + jndi;
    }
}
```

Applications that utilize Kodo can then obtain a handle to the `PersistenceManagerFactory` as follows:

Example 10.2. Looking up the `PersistenceManagerFactory` in JNDI

```
PersistenceManagerFactory factory = (PersistenceManagerFactory)
    new InitialContext ().lookup ("java:/MyKodoJNDIName");
PersistenceManager pm = factory.getPersistenceManager ();
```

The third way to integrate Kodo in an application server is to utilize the Java Connector Architecture features of Kodo as described below.

10.2. Using Kodo JDO via the Java Connector Architecture

10.2.1. Overview of the JCA

An introduction to JCA in Javaworld gives a good overview of the Java Connection Architecture:

The JCA defines a standard set of interfaces that allows connectors to integrate with compliant application servers seamlessly. At the same time, another standard set of interfaces allows clients (or applications hosted by the application server) to use the connectors in a uniform way. Thus with JCA, connectors are portable across application servers, and clients are portable across connectors.

—Tarak Modi

10.2.2. Deploying on JBoss 3.0

JBoss 3.0 will automatically deploy the `kodo.rar` file (located in this distribution's `JCA/` directory) when it is placed in JBoss' `deploy/` directory. However, you will need to configure the parameters of the Resource Adaptor by creating a `kodo-service.xml` file in the JBoss `deploy/` directory. A template `kodo-service.xml` is provided with this distribution in the `JCA/` directory.

Note that while JBoss versions lower than 3.0 support the JCA, they provide no way to configure the `ManagedConnectionFactoryProperties`, which is the primary means of configuring Kodo as a service in a managed environment. If you want to deploy Kodo to JBoss 2.4.4 or lower, you will need to extract the `ra.xml` from the `kodo.rar` file, change the default values settings to match your configuration, and then repackage the files back into the `kodo.rar` file.

To deploy to JBoss, you must do the following:

- Copy `JCA/kodo.rar` to the `deploy/` directory of your JBoss installation.
- Edit `JCA/kodo-service.xml` to set up the configuration for your environment. In particular, you must set the `LicenseKey` property to your license key, and you will probably need to change the `ConnectionDriverName`, `ConnectionURL`, `ConnectionUserName`, `ConnectionPassword`, and `DictionaryName` properties.
- Copy the updated `JCA/kodo-service.xml` to the `deploy/` directory of your JBoss installation.

Chapter 11. Kodo JDO Implementation Notes

This chapter discusses implementation-specific details that may be important to developers.

11.1. PersistenceCapable.jdoFlags field and fields in the Default Fetch Group

When loading the default fetch group fields into a `PersistenceCapable` object that is not involved in a transaction, Kodo JDO will set the `jdoFlags` field of the object to `READ_OK`. This means that subsequent read operations will not be mediated by the `StateManager`, reducing the number of method calls and therefore increasing performance.

When loading the default fetch group fields into a `PersistenceCapable` object that is involved in a transaction, the `jdoFlags` field of the object to `READ_WRITE_OK`. This means that subsequent write operations to fields in the default fetch group will not perform checks against the data store, or otherwise access the `StateManager`. Additionally, when writing the data back out to the data store, all fields in the default fetch group will be written. This is because Kodo JDO does not intercept writes to fields in the default fetch group after they have been loaded, so it is impossible to know which of these fields have been modified. This may result in undesirable data store behavior, including unexpected firing of triggers. Explicitly removing elements from the default fetch group will resolve this.

See section 20.9.3 of the JDO specification for more details about this behavior.

11.2. Optimistic locking mechanism

When the `javax.jdo.option.Optimistic` option is enabled, or optimistic locking is otherwise turned on, Kodo will utilize the value of the `lock-column` metadata extension (which default to `JDOLOCKXX`) to determine which column to use for its "lock column". The lock column holds an integer which is incremented each time changes are committed to the object, and is utilized to determine whether an optimistic transaction should succeed or fail.

Chapter 12. Optimization Techniques

There are numerous techniques the developer can use in order to ensure that Kodo operates in the fastest and most efficient manner. Following is a list of guidelines, in the approximate order of importance:

1. *Database indices*: Indices created by Kodo's schematool may not always be the more appropriate for your application. Manually manipulating indices to include frequently-queried fields (as well as dropping indices on rarely-queried fields) can yield significant performance benefits.
2. *Use the best JDBC Driver*: The JDBC driver provided by the database vendor is not always the fastest and most efficient. Some JDBC drivers do not support features like batched statements, the lack of which can significantly slow down Kodo's data access.
3. *JVM optimizations*: Manipulating various parameters of the Java Virtual Machine (such as hotspot compilation modes and the maximum memory) can result in performance improvements. For more details about optimizing the JVM execution environment, please see <http://java.sun.com/docs/hotspot/PerformanceFAQ.html>.
4. *Use the data cache*: Using the Kodo Data Cache feature (available as a separate product) can often result in a dramatic improvement in performance. See the DataStore Cache chapter for more details.
5. *Disable logging, performance tracking, SynchronizeSchema*: Developer options such as logging, the PerformanceTracker, and using the SynchronizeSchema developer option will result in serious performance hits for your application. Before evaluating any Kodo performance, these options should all be disabled.
6. *Ensure that batch updates are available*: When performing bulk inserts, updates, or delete, Kodo will use batched statements. If this feature is not available in your JDBC driver, then Kodo will need to issue multiple SQL statements instead of a single batch statement.
7. *Use single-table inheritance*: Using a single-table inheritance model is faster for most operations than a multi-table inheritance model. If it is appropriate for your application, you should use the single-table inheritance model whenever possible.
8. *Use unordered Sets instead of Lists*: There is extra overhead for Kodo to maintain ordered Collections (either as relations or privately-owned Collections). If your application does not require ordering for a relation or Collection, you should always use a HashSet as opposed to a LinkedList, ArrayList, or SortedSet.
9. *High increment in DBSequenceFactory*: For applications that perform large bulk inserts, a bottleneck can be the retrieval of sequence numbers. Incrementing the value of the Increment parameter of the `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties` property can result in this bottleneck being reduced. In some cases, implementing your own SequenceFactory can be used to optimize sequence number retrieval.
10. *Use optimistic transactions*: Using datastore transactions translates into pessimistic database row locking, which can be a performance hit (depending on the database). If appropriate for your application, using optimistic transactions are typically faster than using datastore transactions.
11. *Perform nontransactional data reads outside a transaction*.
12. *Always close PersistenceManagers and Query results*: It is important to bear in mind that a PersistenceManager and the result from a Query are often backed by resources in the database. For example, if a Query result has not been completely instantiated, it will hold open a ResultSet object, which, in turn, will hold open a Statement object (preventing it from being re-used). Garbage collection will clean up these resources, so it is never necessary to explicitly close these resources, but it is always faster if it is done at the application level.

Example 12.1. Explicitly closing resources

```

public int getPersonCount (String jdoql)
{
    PersistenceManagerFactory factory = ...; // obtain a PersistenceManagerFactory
    PersistenceManager pm = factory.getPersistenceManager ();
    try
    {
        Query query = pm.newQuery (Person.class, jdoql);
        try
        {
            return ((Collection)query.execute ()).size ();
        }
        finally
        {
            // close all results from this query
            query.closeAll ();
        }
    }
    finally
    {
        // close the PersistenceManager and any associated resources
        pm.close ();
    }
}

```

13. *Optimize connection pool settings:* Kodo's built-in connection pool's default settings may not be optimal for all applications. For applications that instantiate and close many PersistenceManagers (such as a web application), increasing the size of the connection pool will reduce the overhead of waiting on free connections or opening new connections.
14. *Utilize the PM cache:* When possible and appropriate, re-using PersistenceManager objects will result in huge performance gains, since the PersistenceManager's built-in object cache will be used.
15. *Enable Multithreaded operation only when necessary:* Kodo respects the `javax.jdo.option.Multithreaded` option in that it does not impose synchronization overhead for applications that set this value to false. If your application is guaranteed to be single-threaded, setting this option to false will result in the elimination of synchronization overhead, and may result in a modest performance increase.
16. *Disable large data set handling:* By default, Kodo JDO creates statements with the `ResultSet.TYPE_SCROLL_INSENSITIVE` flag. On some databases (SQLServer for example), result sets that support bidirectional scrolling are much slower than unidirectional result sets. So, if you do not have lots of data or your application always fully traverses large data sets, then you should disable large data set handling by setting the `DefaultFetchThreshold` property to -1. A future release of Kodo JDO will allow unidirectional lazily loaded result lists, which will permit lazy materialization of large result sets without incurring the penalty of using `TYPE_SCROLL_INSENSITIVE` result sets.
17. *Develop a custom SubclassProvider, use the `com.solarmetric.kodo.impl.jdbc.ormapping.IntegerSubclassProvider`, or turn off the subclass indicator column.* Kodo JDO's default subclass provider is quite robust, in that it can handle any class and needs no configuration, but the downside of this robustness is that it puts a relatively lengthy string into each row of the database. With the `IntegerSubclassProvider` provider, a little application-specific configuration, you could easily reduce this to an integer. This can result in significant performance gains when dealing with many small objects, since the subclass indicator data can become a significant proportion of the data transferred between the JVM and the database.

Alternately, if your application does not make use of inheritance, then you can disable the subclass provider column altogether.

18. *Set the `com.solarmetric.kodo.CacheReferenceSize` property to -1:* Setting this property to -1 will cause the PersistenceManager to maintain hard references to all objects loaded through that PM, causing

potential memory issues. However, it will no longer be necessary to maintain any ordering in this cache, so systems that load lots of objects may see some performance improvements.

Part IV. Kodo JDO Tutorial

Introduction to the Kodo JDO Tutorial

This tutorial provides a step-by-step example of how to use the Kodo JDO system. It assumes a general knowledge of JDO and Java. For more information on these subjects, see the following URLs:

- [Sun's Java site](#)
- [JDO JSR page](#)
- [JDO Overview Document](#)
- [Locally mirrored javax.jdo Javadoc](#)

This tutorial also assumes some familiarity with the concepts covered in the Kodo JDO Reference Guide.

1.1. Tutorial Requirements

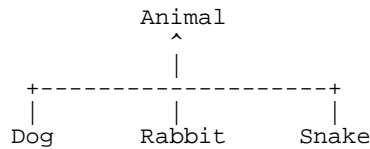
This tutorial requires that JDK 1.2 or greater be installed on your computer, and that 'java' and 'javac' are in your path when you open a command shell. See README.txt for more information on requirements and installation procedures.

Chapter 1. The Pet Shop

Imagine that you have decided to create a software toolkit to be used by pet shop operators. This toolkit must provide a number of solutions to common problems encountered at pet shops. Industry analysts indicate that the three most desired features are inventory maintenance, inventory growth simulation, and behavioral analysis. Not one to question the sage advice of experts, you choose to attack these three problems first.

According to the aforementioned experts, most pet shops focus on three types of animals only: dogs, rabbits, and snakes. This ontology suggests the following class hierarchy:

Class Hierarchy



1.1. Included Files

We have provided an implementation of `Animal` and `Dog` classes, plus some helper classes and files to create the initial schema and populate the database with some sample dogs. Let's take a closer look at these classes.

- `tutorial.JDOFactory` provides access to a properly configured `javax.jdo.PersistenceManagerFactory`.

The `PersistenceManagerFactory` class is the main entry point into the JDO framework. `PersistenceManagerFactories` provide a mechanism for obtaining individual `PersistenceManager` instances, through which a user can find objects in the underlying data store. See the `javax.jdo` javadoc for a more in-depth discussion of JDO in general and the `PersistenceManagerFactory` class in particular.

- `tutorial.AnimalMaintenance` provides some utility methods for examining and manipulating the animals stored in the database. We will fill in method definitions in Chapter III.
- `tutorial.Animal` is the superclass of all animals that this pet store software can handle.
- `tutorial.Dog` contains data and methods specific to dogs.
- `tutorial.Rabbit` contains data and methods specific to rabbits. It will be used in Chapter IV of this tutorial.
- `tutorial.Snake` contains data and methods specific to snakes. It will be used in Chapter V of this tutorial.
- `tutorial.jdo` is a JDO metadata file that defines which types should be enhanced into persistence-capable or persistence-aware classes. For more information on JDO metadata, consult the metadata section of the JDO Overview.
- `kodo.properties` is a properties file containing Kodo-specific and standard JDO configuration settings.

Important

You must specify a valid Kodo license key in the `kodo.properties` file.

- `solutions` contains the complete solutions to this tutorial, including finished versions of the `.java` files listed above and a correct `tutorial.jdo` metadata file.

1.2. Important Utilities

- **java** runs main methods in specified Java classes.
- **javac** compiles .java files into .class files that can be executed by **java**.
- **jdoc** or **java com.solarmetric.kodo.enhance.JDOEnhancer** runs the Kodo JDO enhancer against the specified classes. More information is available in the enhancer section of the Reference Guide.
- **schematool -action refresh** or **java com.solarmetric.kodo.impl.jdbc.schema.SchemaTool -action refresh** is a SolarMetric-specific utility class that can be used to create and maintain a schema for all persistent classes in a JDBC-compliant data store. This functionality allows the underlying schema to be easily kept up-to-date with the Java classes in the system. See the schema section of the Reference Guide for more information.

Chapter 2. Getting Started

Let's compile the initial classes and see them in action. To do so, we must compile the .java files, as we would with any Java project, and then pass the resulting classes through the JDO enhancer:

1. Change to the `tutorial` directory

All examples throughout the tutorial assume that you are in this directory.

2. Examine `Animal.java`, `Dog.java`, and `SeedDatabase.java`

These files are good examples of the simplicity JDO engenders. As noted earlier, persisting an object or manipulating an object's persistent data requires almost no JDO specific code. For a very simple example of creating persistent objects, please see the main method of `SeedDatabase.java`. Note the objects are created with normal Java constructors. The files `Animal.java` and `Dog.java` are also good examples of how JDO allows you to manipulate persistent data without writing any specific JDO code.

3. Compile the .java files

javac *.java

You can use any java compiler instead of **javac**.

4. Enhance the JDO classes

jdoc tutorial.jdo (or **java com.solarmetric.kodo.enhance.JDOEnhancer tutorial.jdo**)

This step runs the Kodo JDO enhancer on the `tutorial.jdo` file mentioned above. The `tutorial.jdo` file contains an enumeration of all the classes that should be JDO enhanced. The Kodo JDO enhancer will examine the contents of this file and enhance all classes listed in it according to the metadata defined in the file. See the enhancer section of the Reference Guide for more information on the JDO enhancer.

Configure the data store

Now that we've compiled the source files and enhanced the JDO classes, we're ready to set up the database. Included in this distribution is Hypersonic SQL, a pure Java relational database. We have included this database because it is simple to set up and has a small memory footprint; however, you can use this tutorial with any of the relational databases that we support. You can also write your own plugin for any database that we do not support. For the sake of simplicity, this tutorial describes how to set up connectivity only to a Hypersonic SQL database. For more information on how to connect to a different database or how to add support for other databases, see the database section of the Reference Guide.

1. Create the database

schematool -action refresh tutorial.jdo or **java com.solarmetric.kodo.impl.jdbc.schema.SchemaTool -action refresh tutorial.jdo**

This command refreshes the database configured in `kodo.properties` with the classes listed in `tutorial.jdo`. If you are using the default Hypersonic SQL setup, the first time you run the schema tool Hypersonic will create `tutorial_database.properties` and `tutorial_database.script` database files in your current directory. To delete the database, just delete these files.

2. Populate with sample data

java tutorial.SeedDatabase

This and all other commands except **schematool** output all SQL statements sent to the JDBC driver to

`./sql.txt`. The logging configuration is specified by the `com.solarmetric.kodo.Logger` property in `kodo.properties`. Change the value of this property to `stdout` to log to standard output instead. Delete the property to disable logging.

Congratulations! You have now created a JDO-accessible persistent store, and seeded it with some sample data.

Chapter 3. Inventory Maintenance

The most important element of a successful pet store product, say the experts, is an inventory maintenance mechanism. So, let's work on the `Animal` and `Dog` classes a bit to permit user interaction with the database.

This chapter should familiarize you with some of the basics of the JDO spec and the mechanics of compiling and enhancing persistence-capable objects. You will also become familiar with the `SchemaTool` for propagating the JDO schema into the database.

First, let's add some code to `AnimalMaintenance.java` that allows us to examine the animals currently in the database.

1. Add code to `AnimalMaintenance.java`

Modify the `getAnimals` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Return a collection of animals that match the specified query filter.
 *
 * @param filter    the JDO filter to apply to the query
 * @param cls       the class of animal to query on
 * @param pm        the PersistenceManager to obtain the query from
 */
public static Collection getAnimals (String filter, Class cls,
    PersistenceManager pm)
{
    // Get a query for the specified class and filter.
    Query query = pm.newQuery (cls, filter);

    // Add a single variable of type 'Animal' to this query to allow
    // for some reasonably powerful queries. This will be uncommented
    // in Chapter V.
    // query.declareVariables ("Animal animal;");

    // Execute the query.
    return (Collection) query.execute ();
}
```

2. Compile `AnimalMaintenance.java`

```
javac AnimalMaintenance.java
```

3. Take a look at the animals in the database

```
java tutorial.AnimalMaintenance list Animal
```

Notice that `list` optionally takes a query filter. Let's explore the database some more, this time using filters:

```
java tutorial.AnimalMaintenance list Animal "name == \"Binney\""
```

```
java tutorial.AnimalMaintenance list Animal "price <= 50"
```

The JDO query language is designed to look and behave much like boolean expressions in Java. The name and price fields identified in the above queries map to the member fields of those names in `tutorial.Animal`. More details on JDO query syntax is available in the JDO Overview. For a definitive reference, consult the JDO specification.

Great! Now that we can see the contents of the database, let's add some code that lets us add and remove animals.

Adding dogs to the database

As new dogs are born or acquired, the store owner will need to add new records to the inventory database. In this

section, we'll write the code to handle additions through the `tutorial.AnimalMaintenance` class.

This section will familiarize you with the mechanism for storing persistence-capable objects in a JDO persistence manager. We will create a new dog, obtain a `Transaction` from a `PersistenceManager`, and, within the transaction, make the new dog object persistent in our `PersistenceManager`.

`tutorial.AnimalMaintenance` provides a reflection-based facility for creating any type of animal, provided that the animal has a two-argument constructor whose first argument corresponds to the name of the animal to add and whose second argument is an implementation-specific primitive. This reflection-based system is in place to keep this tutorial short and remove repetitive creation mechanisms. It is not a required part of the JDO specification.

1. Add the following code to `AnimalMaintenance.java`

Modify the `persistObject` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Performs the actual JDO work of putting <code>object</code>
 * into the data store.
 *
 * @param object the object to persist in the data store
 */
public static void persistObject (Object object)
{
    // Get a PersistenceManager from the JDOFactory.
    PersistenceManager pm = JDOFactory.getPersistenceManager ();

    // Obtain a transaction and mark the beginning
    // of the unit of work boundary.
    Transaction transaction = pm.currentTransaction ();
    transaction.begin ();

    pm.makePersistent (object);

    // Mark the end of the unit of work boundary,
    // and record all inserts in the database.
    transaction.commit ();

    System.out.println ("Added " + object);

    // Close the PersistenceManager.
    pm.close ();
}
```

2. Recompile `AnimalMaintenance.java`

`javac AnimalMaintenance.java`

You now have a mechanism for adding new dogs to the database. Go ahead and add some by running **`java tutorial.AnimalMaintenance add Dog <name> <price>`** (for example, **`java tutorial.AnimalMaintenance add Dog Fluffy 35`**), and look at the new contents of the database with **`java tutorial.AnimalMaintenance list Dog`**.

Removing animals from the database

What if someone decides to buy one of the dogs? The store owner will need to remove that animal from the database, since it is no longer in the inventory.

This section demonstrates how to remove data from the data store.

1. Add the following code to `AnimalMaintenance.java`

Modify the `deleteObjects` method of `AnimalMaintenance.java` to look like this:

```

/**
 * Performs the actual JDO work of removing <code>objects</code>
 * from the data store.
 *
 * @param objects    the objects to persist in the data store
 * @param pm         the PersistenceManager to obtain the query from
 */
public static void deleteObjects (Collection objects, PersistenceManager pm)
{
    // Obtain a transaction and mark the beginning of the
    // unit of work boundary.
    Transaction transaction = pm.currentTransaction ();
    transaction.begin ();

    for (Iterator iter = objects.iterator (); iter.hasNext (); )
    {
        System.out.println ("Removed animal: " + iter.next ());
    }

    // This method removes the objects in 'objects' from the
    // data store.
    pm.deletePersistentAll (objects);

    // Mark the end of the unit of work boundary, and record all
    // deletes in the database.
    transaction.commit ();
}

```

2. Recompile `AnimalMaintenance.java`

javac AnimalMaintenance.java

3. Remove some animals from the database

java tutorial.AnimalMaintenance remove Animal <query>, where *<query>* is a query string like those used for listing animals above.

All right. We now have a basic pet shop inventory management system. From this base, we will add some of the more advanced features suggested by our industry experts.

Chapter 4. Inventory Growth

Now that we have the basic pet store framework in place, let's add support for the next pet in our list: the rabbit. The rabbit is a bit different than the dog; pet stores sell them all for the same price, but gender is critically important since rabbits reproduce rather easily and quickly. Let's put together a class representing a rabbit.

In this chapter, you will see some more queries and write a two-sided many-to-many relation between objects.

Provided with this tutorial is a file called `Rabbit.java` which contains a sample `Rabbit` implementation. Let's get it compiled and loaded:

1. Examine and Compile `Rabbit.java`

javac Rabbit.java

2. Add an entry for `Rabbit` to `tutorial.jdo`

The `Rabbit` class above contains a many-to-many relationship, between parents and children. From the Java side of things, a JDO many-to-many relationship is simply a pair of collections that are conceptually linked. There is no special Java work necessary to express a relationship. However, you must identify the relationship in the JDO metadata for the class. The snippet below should be inserted into the `tutorial.jdo` file. It identifies both the type of data in the collection (the 'element-type' attribute) and the name of the other side of the relation. Notice the use of the 'extension' element. Since the concept of a two-sided relationship is not a data store-independent concept, the JDO specification does not provide built-in support for identifying the inverse of a relation. With this in mind, SolarMetric has used the JDO extension mechanism to add inverse metadata to the JDO metadata file. For more information on metadata, consult the metadata section of the Kodo JDO Reference Guide.

Add the following code on the line immediately *before* the "`</package>`" line in the `tutorial.jdo` file.

```
<class name="Rabbit" persistence-capable-superclass="Animal" >
  <field name="parents">
    <collection element-type="Rabbit"/>
    <extension vendor-name="kodo" key="inverse" value="children"/>
  </field>
  <field name="children">
    <collection element-type="Rabbit"/>
    <extension vendor-name="kodo" key="inverse" value="parents"/>
  </field>
</class>
```

3. Enhance the `Rabbit` class

jdoc tutorial.jdo (or jdoc Rabbit.class)

4. Refresh the database

schematool -action refresh tutorial.jdo (or schematool -action refresh Rabbit.class)

Now that we have a `Rabbit` class, let's get some preliminary rabbit data into the database.

1. Create some rabbits

Run **java tutorial.AnimalMaintenance add Rabbit <name> false** and **java tutorial.AnimalMaintenance add Rabbit <name> true** a few times to add some male and female rabbits to the database; then run **java tutorial.Rabbit breed 2** to run some breeding iterations.

2. Look at your new rabbits

Run **java tutorial.AnimalMaintenance list Rabbit** and **java tutorial.AnimalMaintenance details Rabbit** to look at the contents of the database.

Chapter 5. Behavioral Analysis

Often, pet stores sell snakes as well as rabbits and dogs. Pet stores are primarily concerned with a snake's length; much like rabbits, pet store operators usually sell them all for a flat rate.

This chapter demonstrates more queries, schema manipulation, one-to-many relations, and many-to-many relations.

Provided with this tutorial is a file called `Snake.java` which contains a sample Snake implementation. Let's get it compiled and loaded:

1. Examine and Compile `Snake.java`

javac Snake.java

2. Add `tutorial.Snake` to `tutorial.jdo`

```
<class name="Snake" persistence-capable-superclass="Animal" />
```

3. Enhance the class

jdoc tutorial.jdo (or **jdoc Snake.class**)

4. Refresh the database

As we have created a new persistence-capable class, we must change the database schema to match. So run **schematool -action refresh tutorial.jdo** (or **schematool -action refresh Snake.class**) to propagate the changes to the database.

Once you have compiled everything, add a few snakes to the database using **java tutorial.AnimalMaintenance add Snake "name" <length>**, where *<length>* is the length in feet for the new snake. Run **java tutorial.AnimalMaintenance list Snake** to see the new snakes in the database.

Unfortunately for the massively developing rabbit population, snakes often eat rabbits. Any good inventory system should be able to capture this behavior. So, let's add some code to `Snake.java` to support the snake's eating behavior.

First, let's modify `Snake.java` to contain a list of eaten rabbits.

1. Add the following code snippet to `Snake.java`

This is the 'many' side of a one-to-many relation.

```
// *** add this member variable declaration ***
// This list will be persisted into the database as
// a one-to-many relation.
private Set giTract = new HashSet ();

...

// *** modify toString (boolean) to output the giTract list ***
public String toString (boolean detailed)
{
    StringBuffer buf = new StringBuffer (1024);
    buf.append ("Snake ").append (getName ());

    if (detailed)
    {
        buf.append (" (").append (length).append (" feet long) sells for ");
        buf.append (getPrice ()).append (" dollars.");
    }
}
```

```

        buf.append (" Its gastrointestinal tract contains:\n");
        for (Iterator iter = giTract.iterator (); iter.hasNext ();)
        {
            buf.append ("\t").append (iter.next ().append ("\n");
        }
    }
    else
    {
        buf.append ("; ate " + giTract.size () + " rabbits.");
    }
}

return buf.toString ();
}
...

// *** add these methods ***
/**
 * Kills the specified rabbit and eats it.
 */
public void eat (Rabbit dinner)
{
    // Consume the rabbit.
    dinner.kill ();
    dinner.eater = this;
    giTract.add (dinner);
    System.out.println ("Snake " + getName () + " ate rabbit "
        + dinner.getName () + ".");
}

/**
 * Locates the specified snake and tells it to eat a rabbit.
 */
public static void eat (String filter)
{
    PersistenceManager pm = JDOFactory.getPersistenceManager ();

    // Find the desired snake(s) in the data store.
    Query query = pm.newQuery (Snake.class, filter);
    Collection results = (Collection) query.execute ();

    if (results.size () > 0)
    {
        Iterator iter = results.iterator ();
        Query uneatenQuery = pm.newQuery (Rabbit.class, "isDead == false");

        Transaction transaction = pm.currentTransaction ();
        transaction.begin ();

        while (iter.hasNext ())
        {
            // Find a rabbit to eat.
            Random random = new Random ();

            // Run a query for a rabbit whose 'isDead' field indicates
            // that it is alive.
            List menu = new ArrayList ();
            menu.addAll ((Collection) uneatenQuery.execute ());
            if (menu.size () == 0)
            {
                System.out.println ("No live rabbits in DB.");
            }

            // Select a random rabbit from the list.
            Rabbit dinner = (Rabbit) menu.get (random.nextInt
                (menu.size ()));

            // Perform the eating.
            Snake snake = (Snake) iter.next ();

```

```

        System.out.println (snake + " is eating:");
        snake.eat (dinner);
    }

    transaction.commit ();
}
else
{
    System.out.println ("no snakes matching '" + filter
        + "' found in persistence manager");
}
pm.close ();
}

public static void main (String [] args)
{
    if (args.length == 2 && args[0].equals ("eat"))
    {
        eat (args[1]);
        return;
    }

    // If we get here, something went wrong.
    System.out.println ("Usage:");
    System.out.println ("    java tutorial.Snake eat \"snakequery\"");
}

```

2. Add an eater field to Rabbit.java

This is the 'one' side of a one-to-many relation. Notice that we are making this field `protected`. This demonstrates that it is possible to enhance any field; you need not provide getters and setters as we have done in previous examples. This is not recommended practice, because it means that all classes that access this field directly must be JDO enhanced, even if they are not persistence-capable.

Add the following member variable to `Rabbit.java`:

```
protected Snake eater;
```

3. Add metadata to tutorial.jdo

Notice the `giTract` declaration: it is a simple Java list declaration. As with the many-to-many declarations in `Rabbit.java`, we will add metadata to the `tutorial.jdo` file.

```

<class name="Snake" persistence-capable-superclass="Animal" >
  <field name="giTract">
    <collection element-type="Rabbit"/>
    <extension vendor-name="kodo" key="inverse" value="eater"/>
  </field>
</class>

```

Note that we specified an `inverse` attribute in this example. This is because the relation is a two-sided one; that is, the rabbit has knowledge of which snake ate it. We could have left out the `eater` field and instead created a one-sided many-to-many relation. The metadata might have looked like this:

```

<class name="Snake" persistence-capable-superclass="Animal" >
  <field name="giTract">
    <collection element-type="Rabbit"/>
  </field>
</class>

```

For more information on types of relations, see the metadata section of the Kodo JDO Reference Guide.

4. Compile Snake.java and Rabbit.java and jdo-enhance the classes

```
javac Snake.java Rabbit.java
```

```
jdoc tutorial.jdo (or jdoc Snake.class Rabbit.class)
```

5. Refresh the database

```
schematool -action refresh tutorial.jdo
```

Now, experiment with the following classes: **java tutorial.Snake eat** and **java tutorial.AnimalMaintenance details Snake**.

Complex Queries

Imagine that one of the snakes in the database was named 'Killer'. To find out which rabbits Killer ate, we could run either of the following two queries:

- **java tutorial.AnimalMaintenance details Snake "name == \'Killer\'"**
- **java tutorial.AnimalMaintenance list Rabbit "eater.name == \'Killer\'"**

The first query is Snake-centric -- the query runs against the Snake class, looking for all snakes named 'Killer' and providing a detailed listing of them. The second is Rabbit-centric -- it examines the rabbits in the database for instances whose eater field is named 'Killer'. This second query demonstrates the that simple java 'dot' syntax is used when traversing an Object field in a query.

It is also possible to traverse Collection fields. Imagine that there was a rabbit called Roger in the data store and that one of the snakes ate it. In order to determine who ate Roger Rabbit, you could run a query like this:

- **java tutorial.AnimalMaintenance details Snake "giTract.contains (animal) && animal.name == \'Roger\'"**

Note the use of the animal variable that was registered with the query. To add support for this variable, uncomment the commented-out declareVariables call in the getAnimals method in AnimalMaintenance.java:

1. AnimalMaintenance.getAnimals should look like this:

```
/**
 * Return a collection of animals that match the specified query filter.
 *
 * @param filter    the JDO filter to apply to the query
 * @param cls       the class of animal to query on
 * @param pm        the PersistenceManager to obtain the query from
 */
public static Collection getAnimals (String filter, Class cls,
    PersistenceManager pm)
{
    // Get a query for the specified class and filter.
    Query query = pm.newQuery (cls, filter);

    // Add a single variable of type 'Animal' to this query to allow
    // for some reasonably powerful queries. This will be uncommented
    // in Chapter V.
    query.declareVariables ("Animal animal;");

    // Execute the query.
    return (Collection) query.execute ();
}
```

2. Recompile `AnimalMaintenance.java`

`javac AnimalMaintenance.java`

The `animal` variable is now available to use to represent any `Animal` contained in a `Collection`. As `animal` was declared as type `Animal`, we cannot directly use fields in our `Animal` subclasses. However, the JDO specification provides support for casting in queries, which lets us run queries like **`java tutorial.AnimalMaintenance details Snake "giTract.contains (animal) && ((Rabbit)animal).isFemale == false"`**, which prints details about all snakes that have eaten male rabbits.

Chapter 6. Extra Features

Congratulations! You are now the proud author of a pet store inventory suite. Now that you have all the major features of the pet store software implemented, it's time to add some extra features. You're on your own; think of some features that you think a pet store should have, or just explore the features of JDO.

Here are a couple of suggestions to get you started:

- Animal pricing

Modify Animal to contain an inventory cost and a resale price. Calculate the real dollar amount eaten by the snakes (the sum of the inventory costs of all the consumed rabbits), and the cost assuming that all the eaten rabbits would have been sold had they been alive. Ignore the fact that the rabbits, had they lived, would have created more rabbits, and the implications of the reduced food costs due to the not-quite-as-hungry snakes and the smaller number of rabbits.

- Dog categorization

Modify Dog to have a many-to-many relation to a new class called 'Breed', which contains a name identifying the breed of the dog and a description of the breed. Put together an admin tool for breeds and for associating dogs and breeds.

Part V. Reverse Mapping Tool Tutorial

Introduction to the Reverse Mapping Tool Tutorial

This tutorial provides a step-by-step example of how to use Kodo JDO's reverse mapping tool to reverse-engineer persistent classes from a database schema. It assumes a general knowledge of JDO and Java. For more information on these subjects, see the following URLs:

- Sun's Java site
- Sun's JDO site
- JDO Overview Document
- Locally mirrored javax.jdo Javadoc

This tutorial also assumes some familiarity with the concepts covered in the Kodo JDO Reference Guide.

1.1. Reverse Mapping Tool Tutorial Requirements

This tutorial requires that JDK 1.2 or greater be installed on your computer, and that 'java' and 'javac' are in your path when you open a command shell. See README.txt for more information on requirements and installation procedures.

Chapter 1. Magazine Shop

You run a shop that sells magazines. You store information about your inventory in a relational database with the following schema:

```
-- Holds information on available magazines
CREATE TABLE MAGAZINE (
    ISBN VARCHAR(8) NOT NULL,
    ISSUE INTEGER NOT NULL,
    NAME VARCHAR(255),
    PUBLISHER_NAME VARCHAR(255)
    PRICE FLOAT NOT NULL,
    PRIMARY KEY (ISBN, ISSUE)
    FOREIGN KEY (PUBLISHER_NAME) REFERENCES PUBLISHER (NAME)
);

-- Holds information on magazine articles
CREATE TABLE ARTICLE (
    TITLE VARCHAR(255) NOT NULL,
    AUTHOR_NAME VARCHAR(128),
    PRIMARY KEY (TITLE)
);

-- Holds all the subtitles of an article
CREATE TABLE ARTICLE_SUBTITLES (
    ARTICLE_TITLE VARCHAR(255),
    SUBTITLE VARCHAR(128),
    FOREIGN KEY (ARTICLE_TITLE) REFERENCES ARTICLE (TITLE)
);

-- Join table linking magazines and articles
CREATE TABLE MAGAZINE_ARTICLES (
    MAGAZINE_ISBN VARCHAR(8),
    MAGAZINE_ISSUE INTEGER,
    ARTICLE_TITLE VARCHAR(255),
    FOREIGN KEY (MAGAZINE_ISBN, MAGAZINE_ISSUE) REFERENCES MAGAZINE (ISBN, ISSUE),
    FOREIGN KEY (ARTICLE_TITLE) REFERENCES ARTICLE (TITLE)
);

-- Each magazine has a 1-1 relation to its publisher
CREATE TABLE PUBLISHER (
    NAME VARCHAR(255) NOT NULL,
    REVENUE FLOAT NOT NULL,
    PRIMARY KEY (NAME)
);
```

You've decided to write an application that will let you query your database through JDO.

Chapter 2. Setup

If you haven't already, follow the instructions in the Kodo JDO README. This will ensure that your `CLASSPATH` and other environmental variables are set up correctly. Once you've completed the installation instructions, change into the `reversetutorial` directory.

2.1. Tutorial Files

The tutorial uses the following files:

- The `hsql_sample_database.properties`, `hsql_sample_database.script`, and `hsql_sample_database.data` files make up a Hypersonic SQL file-based database with the schema outlined above. The database is already populated with lots of magazine data representing your shop's inventory.
- `reversetutorial.JDOFactory` provides access to a properly configured `javax.jdo.PersistenceManagerFactory`.

The `PersistenceManagerFactory` class is the main entry point into the JDO framework. `PersistenceManagerFactories` provide a mechanism for obtaining individual `PersistenceManager` instances, through which a user can find objects in the underlying data store. See the `javax.jdo` javadoc for a more in-depth discussion of JDO in general and the `PersistenceManagerFactory` class in particular.

- `reversetutorial.Finder` uses JDO to execute user-entered query strings and output the matching persistent objects. This class relies on persistent classes that we haven't generated yet, so it won't compile immediately.
- `kodo.properties` is a properties file containing Kodo-specific and standard JDO configuration settings.

Note that if you've changed your `kodo.properties` settings to access another database, you'll have to change them back for this tutorial. Reset the following properties:

```
javax.jdo.option.ConnectionDriverName=org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionUserName=sa
javax.jdo.option.ConnectionPassword=
javax.jdo.option.ConnectionURL=jdbc:hsqldb:hsql_sample_database
```

You might also want to change switch SQL logging from standard output to a file for the purposes of this tutorial. Change the `Logger` property to look like this:

```
com.solarmetric.kodo.Logger=sql.txt
```

Important

You must specify a valid Kodo license key in the `kodo.properties` file.

- `solutions` contains the complete solutions to this tutorial, including all generated code.

2.2. Important Utilities

- `java` runs main methods in specified Java classes.

- **javac** compiles .java files into .class files that can be executed by **java**.
- **jdoc** or **java com.solarmetric.kodo.enhance.JDOEnhancer** runs the Kodo JDO enhancer against the specified classes. More information is available in the enhancer section of the Reference Guide.

Chapter 3. Generating Persistent Classes

Now it's time to turn your magazine database into persistent JDO classes mapped to each existing table. To accomplish this, we'll use Kodo JDO's reverse schema mapping tools.

Note

Because reverse schema mapping comes from Solarmetric's R&D codebase, the tool scripts all have the `rd-` prefix. The R&D codebase also uses the same Apache commons logging framework used in Kodo JDO 2.4 beta and above. See the commons logging API if you need to configure logging.

1. First, make sure that you are in the `reversetutorial` directory and that you've made the appropriate modifications to your `kodo.properties` file, as described in the previous chapter.
2. Now that we've got our environment set up correctly, we're going to dump our existing schema to an XML document. This step is not strictly necessary for Hypersonic SQL, which provides good database metadata. Some databases, however, have faulty JDBC drivers, and Kodo JDO is unable to gather enough information about the existing schema to create a good object model from it. In these cases, it is useful to dump the schema to XML, then modify the XML by hand to correct any errors introduced by the JDBC driver. If your schema doesn't use foreign key constraints, you may also want to add logical foreign keys to the XML file so that Kodo JDO can create the corresponding relations between the persistent classes it generates.

To perform the schema-to-XML conversion, we're going to use the can be invoked via the included `rd-schemagen` shell script, or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.schema.SchemaGenerator`. The `-file` flag tells the generator what to name the XML file it creates:

```
rd-schemagen -file schema.xml
or
java com.solarmetric.rd.kodo.impl.jdbc.schema.SchemaGenerator -file schema.xml
```

3. Examine the `schema.xml` XML file created by the schema generator. As you can see, it contains a complete representation of the schema for your magazine database.
4. Run the reverse mapping tool on the schema file. (If you do not supply the schema file to reverse map, the tool will run directly against the schema in the database). The tool can be run via the included `rd-reversemappingtool` script, or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.meta.ReverseMappingTool`. Use the `-package` flag to control the package of the generated classes.

```
rd-reversemappingtool -package reversetutorial schema.xml
or
java com.solarmetric.rd.kodo.impl.jdbc.meta.ReverseMappingTool -package reversetutorial schema.xml
```

Running the tool will generate `.java` files for each generated class, `.java` files for corresponding JDO application identity classes, a `reversemapping.jdo` JDO metadata file, a `reversetutorial.schema` file, and a `reversetutorial.mapping` file. The schema file is there to show you what portion of the schema was mapped; it serves no other purpose and can be deleted if desired. The mapping file contains the O/R information mapping the generated classes to your existing schema. Mapping files like these will be offered as an optional alternative to Kodo JDO's standard JDO metadata extensions in a future version of Kodo JDO. For now, you must import the mapping information into Kodo JDO metadata extensions.

5. To import the generated O/R mapping information into metadata extensions, run the R&D import tool on the mapping file. The tool can be invoked via the included `rd-importtool` script or through its Java class, `com.solarmetric.rd.kodo.impl.jdbc.meta.compat.ImportTool`. Make sure to compile

the generated classes before the import:

```
javac *.java
rd-importtool revesetutorial.mapping
or
java com.solarmetric.rd.kodo.impl.jdbc.meta.compat.ImportTool revesetutorial.mapping
```

You can now delete the mapping file if desired.

6. Examine the generated persistent classes. Notice that the reverse mapping tool has used column and foreign key data to create the appropriate persistent fields and relations between classes. Notice, too, that due to the transparency of JDO, the generated code is vanilla Java, with no trace of JDO-specific functionality.

Also examine the generated application identity classes. Note that they satisfy all of the requirements for application identity classes mandated by the JDO specification, including the `equals` and `hashCode` contracts.

Finally, examine the `revesetutorial.jdo` metadata file. It contains the necessary standard JDO metadata, plus the necessary Kodo JDO extensions to map the classes and their fields to the existing schema.

The reverse mapping tool has now created a complete JDO object model for your magazine shop's existing relational model. From now on, you can treat the generated JDO classes just like any other JDO class. And that means you have to complete two additional steps before you can use the classes with persistence.

- Enhance the JDO classes.

```
jdoc revesetutorial.jdo
or
java com.solarmetric.kodo.enhance.JDOEnhancer revesetutorial.jdo
```

This step runs the Kodo JDO enhancer on the `revesetutorial.jdo` file mentioned above. The `revesetutorial.jdo` file contains an enumeration of all the classes that should be JDO enhanced. The Kodo JDO enhancer will examine the contents of this file and enhance all classes listed in it according to the metadata defined in the file. See the enhancer section of the Reference Guide for more information on the JDO enhancer.

- Register the persistent classes. If you are using Kodo JDO version 2.3.x or below, you need to register all persistent classes with Kodo JDO. The easiest way to do this is with the schema tool:

```
schematool -action register revesetutorial.jdo
or
java com.solarmetric.kodo.impl.jdbc.schema.SchemaTool -action register revesetutorial.jdo
```

The schema tool creates a special table in the database to hold the names of registered classes.

Congratulations! You are now ready to use JDO to access your magazine data.

Chapter 4. Using the Finder

The `reversetutorial.Finder` class lets you run queries in JDOQL (JDO's Java-centric query syntax) against the existing database:

```
java reversetutorial.Finder <jdoql-query>
```

JDOQL looks exactly like Java boolean expressions. To find magazines matching a set of criteria in JDOQL, just specify conditions on the `reversetutorial.Magazine` class' persistent fields. Some examples of valid JDOQL queries for magazines include:

```
java reversetutorial.Finder "true" // use this to list all the magazines
java reversetutorial.Finder "price < 5.0"
java reversetutorial.Finder "name == \"Vogue\" || issue > 1000"
java reversetutorial.Finder "name.startsWith (\"V\")"
```

To traverse object relations, just use Java's dot syntax:

```
java reversetutorial.Finder "publisher.name == \"Adventure\" || publisher.revenue > 1000000"
```

To traverse collection relations, you have to use JDOQL variables. Variables are just placeholders for any member of a collection. For example, in the following query, `art` is a variable:

```
java reversetutorial.Finder "articles.contains (art) && art.title.startsWith (\"JDO\")"
```

The above query is equivalent to "find all magazines who have an article whose title starts with 'JDO'". The `reversetutorial.Finder` pre-defines two variables you can use: `Article art;` `Magazine mag;`. With these, you can create very complex queries. For example, to find all magazines whose publisher published an article about "Surpassing Hubble" in any of its magazines:

```
java reversetutorial.Finder "publisher.magazines.contains (mag) && mag.articles.contains (art) && ar
```

Have fun experimenting with additional queries.

Appendix A. Development and Runtime Libraries

-
- `kodo-jdo.jar`: Library for development of applications for Kodo JDO.
- `kodo-jdo-runtime.jar`: Library for runtime use of Kodo JDO.
- `kodo-reverse-schema.jar`: Library that enables the Kodo Reverse Schema tool.
- `jdo1_0.jar`: Interfaces defined by the Java Data Objects standard. Required for all development and runtime use.
- `serp.jar`: Utility classes used internally by Kodo for development and runtime. See <http://serp.sourceforge.net>.
- `jaxp.jar`: Java API for XML Parsing libraries. Used internally by Kodo for parsing `.jdo` metadata files. Required for all development and runtime use of Kodo, unless JDK version 1.4 or higher is being used (in which JAXP is included). See <http://java.sun.com/xml/jaxp/>.
- `xerces.jar`: Apache implementation of the Java API for XML Parsing. Any JAXP-compliant implementation can be used in place of `xerces.jar`. If JDK version 1.4 or higher is being used, this file is optional, since JDK 1.4 includes a built-in JAXP-compliant parser. See <http://xml.apache.org/xerces-j/>.
- `jakarta-commons-logging-1.0.2.jar`: Logging wrapper classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/logging.html>
- `jdbc2_0-stdext.jar`: Standard extensions to the JDBC2 API. Required for all development and runtime use of Kodo, unless JDK version 1.4 or higher is installed (since the library is included with JDK 1.4). See <http://java.sun.com/products/jdbc/>.
- `jca1.0.jar`: Java Connector Architecture, used internally by Kodo. Required for all development and runtime use of Kodo. See <http://java.sun.com/j2ee/connector/>.
- `jdbc-hsql-1_7_0.jar`: Hypersonic pure-java database engine. It is used as a simple JDBC driver for learning Kodo, but it is not required for general development or runtime use. See <http://hsqldb.sourceforge.net/>.

Appendix B. JDO Resources

- [JDO JSR page](#)
- [Kodo JDO community support groups](#)
- [Sun JDO page](#)
- [Locally mirrored javax.jdo javadoc](#)
- [Locally mirrored Kodo javadoc](#)
- [Locally mirrored JDO specification](#)

Appendix C. Optional JDO Features

The following represents a complete list of the optional JDO features supported in Kodo JDO, as well as interesting features specific to relational data stores:

- All JDO lifecycle states are supported.
- All JDO options are supported, with the following exceptions:
 - `javax.jdo.option.ChangeApplicationIdentity`
 - `javax.jdo.option.NullCollection`
- Both datastore and application identity are supported.
- Optimistic transactions, including detection of optimistic locking violations, are supported.
- All Collection types are supported.
- All Map types are supported, including `java.util.Properties`.
- Arrays are supported.
- JNDI storage of `PersistenceManagerFactories` is supported.
- The Kodo JDO `PersistenceManagerFactory` implements the JCA specification. (Enterprise Edition only)
- Query ordering is supported.
- Queries not only support the `contains` method on collection fields, but also the `containsKey` and `containsValue` methods on Map fields.
- `PersistenceManagers` use a soft caching strategy to increase cache hits without overloading JVM memory usage.
- Automatic schema generation and migration for persistent classes is included.
- Automatic generation of identity classes for types using application identity is included.
- Kodo JDO uses 'lazy' database reads for maximum efficiency.
- Kodo JDO supports all standard relational mapping types, including 1-1, 1-M, M-M, Collection, Map, and N-M Map mappings.
- Simple Collection and 1-sided M-M mappings can be configured to maintain order.
- Serialization of fields to BLOB columns is supported.
- Kodo JDO uses a highly customizable inheritance mapping strategy, supporting both single-table and multi-table mappings.
- Large result sets are supported.
- Kodo JDO offers too many opportunities for customization via extension and system plugins to enumerate here.

Appendix D. SQL Types

Following is a table of the java class to SQL type that is generated by the `schematool`.

Note

Kodo does not impose these table constraints: they are merely the default column types that are generated for each of the included dictionaries. These types can easily be overridden by extending `com.solarmetric.kodo.impl.jdbc.schema.dict.GenericDictionary`.

Table D.1. SQL Type Mappings

| | STRING | CLOB | LENSTRING | BOOLEAN | BYTE |
|----------------|---------------|-------------|---------------|-----------|----------|
| MySQL | VARCHAR(255) | TEXT | VARCHAR({0}) | SMALLINT | SMALLINT |
| Hypersonic SQL | VARCHAR(255) | LONGVARCHAR | VARCHAR({0}) | SMALLINT | SMALLINT |
| DB2 | VARCHAR(255) | CLOB(1M) | VARCHAR({0}) | SMALLINT | SMALLINT |
| Generic | VARCHAR(255) | CLOB | VARCHAR({0}) | TINYINT | SMALLINT |
| InstantDB | VARCHAR(255) | TEXT | VARCHAR({0}) | BYTE | BYTE |
| Oracle | VARCHAR2(255) | CLOB | VARCHAR2({0}) | NUMBER(1) | SMALLINT |
| Postgres | VARCHAR(255) | TEXT | VARCHAR({0}) | INT2 | INT2 |
| SQLServer | VARCHAR(255) | TEXT | VARCHAR({0}) | SMALLINT | SMALLINT |

Table D.2. SQL Type Mappings II

| | CHARACTER | REAL | DOUBLE | INTEGER | LONG |
|----------------|-----------|------|----------------|---------|-------------|
| MySQL | CHAR(1) | REAL | DECIMAL(50,80) | INTEGER | BIGINT |
| Hypersonic SQL | CHAR(1) | REAL | DOUBLE | INTEGER | BIGINT |
| DB2 | CHAR(1) | REAL | DOUBLE | INTEGER | BIGINT |
| Generic | CHAR(1) | REAL | DOUBLE | INTEGER | BIGINT |
| InstantDB | CHAR(1) | REAL | DOUBLE | INT | LONG |
| Oracle | CHAR(1) | REAL | NUMBER | NUMBER | NUMBER |
| Postgres | CHAR(1) | REAL | DECIMAL | INT4 | INT8 |
| SQLServer | CHAR(1) | REAL | FLOAT(32) | INTEGER | NUMERIC(19) |

Table D.3. SQL Type Mappings III

| | SHORT | DATE | BLOB | BIGINTEGER | BIGDECIMAL |
|----------------|----------|-----------|-----------|--------------|------------------|
| MySQL | SMALLINT | DATETIME | BLOB | DECIMAL(200) | DECIMAL(200,800) |
| Hypersonic SQL | SMALLINT | TIMESTAMP | VARBINARY | DECIMAL | DECIMAL |
| DB2 | SMALLINT | TIMESTAMP | BLOB(1M) | BIGINT | DOUBLE |
| Generic | SMALLINT | TIMESTAMP | BLOB | BIGINT | DECIMAL |

| | SHORT | DATE | BLOB | BIGINTEGER | BIGDECIMAL |
|-----------|----------|-----------|-----------|-------------------|-------------------|
| InstantDB | SHORT | TIMESTAMP | VARBINARY | NUMERIC(38, 19) | DECIMAL(38, 19) |
| Oracle | NUMBER | DATE | BLOB | NUMBER | NUMBER |
| Postgres | INT2 | TIMESTAMP | BYTEA | NUMERIC | DECIMAL |
| SQLServer | SMALLINT | DATETIME | IMAGE | NUMERIC(38, 0) | NUMERIC(38, 20) |

Appendix E. Release Notes

2.4.3 -- 26 March 2003

- Notable changes

Bugfixes

- Included a new version of `serp` that resolves issues with weak and soft references that can lead to memory leaks. Be sure to copy the new `serp` jar into your `lib` dir as well as the new Kodo jars.
- Fixed a bug in `ClassDBSequenceFactory` to address potential concurrency issues that could lead to a deadlock while obtaining new ID values.
- Fixed `AbstractDictionary` to deal with null `Locale` objects properly.

2.4.2 -- 26 Feb 2003

- Notable changes

- The `ProxyManager` now includes capabilities to proxy user-defined mutable field types that are not part of the JDO specification.
- The SunONE Studio / NetBeans plugin module has moved into release status. Existing module users should un-install and re-install the module.
- The Eclipse / WSAD plugin has moved into release status. *Note that the plugin folder structure has changed to reflect this change.* Existing plugin users should remove the old folder, install the new folder, and update the `plugin.xml` accordingly. Included in this new version are changes in classpath and project resolution.

Bugfixes

- Mutating a `Date` field via deprecated `setDate()`, `setHour()`, etc. now properly dirties the owning object.
- Fixed `LocalCache` synchronization issue.

2.4.1 -- 26 January 2003

- New features

- New subclass provider implementation option simplifies using an integer lookup value to store subclass information in the database. Additionally, the source for this implementation is included in the release, so creating a custom subclass provider is simpler.
- We now set the default transaction isolation level to `TRANSACTION_SERIALIZABLE` when using DB2. This is necessary in order for datastore (pessimistic) transactions to lock rows correctly.
- Added a new `SequenceFactory`: `com.solarmetric.kodo.impl.jdbc.schema.ClassDBSequenceFactory` which provides class sensitive table-based id sequences. To use the new sequence factory, existing sequence tables need to be dropped to be mapped to the differing table structure.

- Added a table name option to `com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory` to map sequences to. To set this option, add the option `tableName=yourname` to `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties` property when configuring your `PersistenceManagerFactory`.
- Persistent types can once again be enumerated by using the `com.solarmetric.kodo.PersistentTypes` property. This property is optional, but help to avoid classloader issues when deploying to an application server.

Bugfixes

- Fixed data caching plug-in to not enlist objects with `can-cache=false` when loading existing data from the database.
- Fixed bug in metadata parsing algorithm that could cause classloader problems in application servers
- Fixed `SQLServerDictionary` to work around `SQLServer`'s issues with setting null BLOBs via `PreparedStatement.setNull()`.
- Datastore locking (i.e., pessimistic locking) is now supported for Sybase. Note that the connection property `"BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true"` must be set, either in the `ConnectionURL` or the `ConnectionProperties` properties. See the `SybaseDictionary.java` source file for more details. This requires the Sybase JDBC driver version 4.5 or higher.

Notable changes

- Removed the `AutoReturnTimeout` property. The Kodo pooling `DataSource` no longer reclaims expired connections.

2.4.0 -- 13 Dec 2002

- New features
 - Pre-release versions of plugins for SunONE Studio, NetBeans, Eclipse, and WebSphere Studio are now available. See the documentation on installation and usage instructions.
 - Kodo JDO Enterprise Edition and the Kodo JDO Performance Pack are now bundled with a cache plug-in that supports Tangosol Coherence cache products. See the datastore cache documentation for details.
 - Added `evictAll(Class)` and `evictAll(Extent)` method calls to `PersistenceManagerImpl`. These methods are useful for clearing often-updated objects in pooled `PersistenceManager` configurations.
 - Added the capability of loading `ResultSet` objects (or any other stream of data) into `PersistenceCapable` objects associated with a `PersistenceManager` via application-defined logic.
 - Added metadata extensions for specifying custom `ClassMapping` and `FieldMapping` values for particular classes and fields.
 - Added class-level metadata extension to exclude certain classes from the `PersistenceManagerFactory` cache.
 - Added property for configuring the how long to wait before testing connections that have been put into the pool.
 - Simplified the process of defining custom subclass indicator behavior.

- When supported by the underlying JDBC driver, Kodo will now use PreparedStatements and batch updates whenever possible for very significant performance benefits. See the documentation for the `com.solarmetric.kodo.impl.jdbc.UsePreparedStatements` and `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements` properties.
- Inverse one-to-one mappings are now supported. The field can now reside on either table corresponding to the related objects. If both sides of an one-to-one are marked as having an inverse, one field should be designated as read-only to indicate to the system the owning class and table for the given relational field.
- Logging is now done through the Apache commons project, which offers the ability to use an underlying logging mechanism, such as Apache Log4J, JDK 1.4's native logging, or simple file/stdout logging. It is now necessary to include the new `jakarta-commons-logging-1.0.2.jar` in the CLASSPATH. See the Logging chapter.
- Added a pluggable `SQLExecutionManager` architecture, which allows the developer to override the mechanism by which SQL is issued to the database. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass` property.
- There is now an option to automatically refresh the database schema during runtime, allowing the developer to skip the `schematool` step. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` property.
- Properties may be specified for a Driver with the `com.solarmetric.kodo.ConnectionProperties` property.
- A `javax.sql.DataSource` may be specified in the `javax.jdo.option.ConnectionDriverName` property, which will be customizable with bean-like entries in the `com.solarmetric.kodo.ConnectionProperties` property.
- The default transaction isolation for a JDBC connection can be overridden with the `com.solarmetric.kodo.impl.jdbc.TransactionIsolation` property.
- Kodo JDO now distributes a single jar for both the enterprise and standard edition, as well as datacache and query extensions.
- The `rd-metadatatool` can now be used to generate default JDO metadata for classes.
- The `rd-schemagen` tool used for reverse mapping classes from a schema has now been tested with the following databases: Hypersonic SQL 7.1, SQLServer (MS Beta 2 JDBC driver), Sybase, Oracle (9.0.1 JDBC driver), DB2, Postgres (7.3 Beta 3 JDBC driver).

Bugfixes

- `default-fetch-group="false"` is now respected for fields that default to `default-fetch-group="true"`.
- Traversing orphaned relations in data cache now behaves in the same way as traversing orphaned db relations -- the invalid relation is set to null.
- Changing a field to the same value as it was originally set to no longer constitutes dirtying that field. This means that subsequent flushes to the database will not necessarily re-write the same data.
- Class loading is performed in accordance with section 12.5 of the JDO specification.

Notable changes

- The API for implementing a `SequenceFactory` has changed. See the API documentation for

`com.solarmetric.kodo.impl.jdbc.SequenceFactory.`

- Persistent types are no longer enumerated, either in the data store or in a property. Classes are now dynamically added upon class initialization, via the `JDOImplHelper` class registration process. The `-register` and `-unregister` options to `schematool` are no longer needed. The `JDO_SCHEMA_METADATA` table is no longer used and can be dropped.

2.3.4

- New features
 - The R&D schema generator can now accept a list of tables to generate.
 - The R&D reverse mapping tool has additional options for using foreign key names to generate relation names, generating primitive wrapper-type fields if a column is nullable, and allowing primary keys on many-to-many join tables.
- Bugfixes
 - Fixed problems with many-to-many relations between tables that use vertical inheritance.
 - Fixed bug in `schematool` that caused it to not generate primary key columns in subclass tables when using datastore identity + custom names + vertical inheritance.
 - Fixed `serp` library conflict between reverse mapping tool and main Kodo libraries.
 - Fixed a reverse mapping tool bug in which column names that conflicted with Java keywords would result in the generation of uncompileable Java classes.
 - Fixed problem that caused read-only flag to be ignored in many-to-many relations.
 - Multi-table inheritance deletes are now performed from the leaf table in the inheritance chain up to the base table. Inserts are performed from the base table down to the leaf. This supports the common referential integrity model of establishing a foreign key relation from inherited tables to their parent tables.

2.3.3

- New features
 - The R&D schema generator can now accept a list of tables to generate.
- Bugfixes
 - Fixed `null-value="default"` behavior.
 - Fixed bugs that prevented removal of map elements through the key set, entry set, and values collection.
 - Added more validation on static/final fields to metadata.
 - Fixed memory leak in `serp` regarding soft and weak collections backed by `HashSets`.
 - Multi-variable query issues resolved.

- Fixed bug that could cause optimistic lock version numbers to be incremented before successful transaction commit.
- The R&D reverse mapping tool now handles Oracle DATE columns correctly.

2.3.2

- New features
 - The new `com.solarmetric.kodo.CacheReferenceSize` property dictates a number of hard references to cached objects that the `PersistenceManager` will retain, in addition to its soft cache.
 - Added 'all' option to unregister action of schematool Ant task. This option allows all classes in the current persistent types list to be unregistered, regardless of whether or not those classes are currently in the classpath.
 - Added `com.solarmetric.kodo.UseSoftTransactionCache` to configure whether or not Kodo should maintain soft references to transactional items that have not been dirtied. This now defaults to false; previous versions of Kodo always used soft references for non-dirty transactional items.
- Bugfixes
 - The `jdodoclet` task no longer creates JDO metadata entries for final or static fields, or for transient fields that do not have a `jdo:persistence-modifier` tag.
 - The `jdoc` task no longer attempts to enhance classes that have already been enhanced.
 - Several default property values were being set improperly.

2.3.1

- New features
 - Smart proxies for set and map fields. Smart proxies better optimize database updates when persistent set and map fields are modified.
 - TCP, JMS-based distributed `DataCache` implementations.
 - All `DataCache` implementations now use an LRU cache with a configurable maximum size.
 - Customizable tracked instance proxies.
 - Alpha release of upcoming reverse mapping tool for creating persistent class definitions, metadata, and mapping extensions from an existing schema.
 - Cache object `com.solarmetric.kodo.runtime.datacache.PMFactoryCache` is now named `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`.
- Bugfixes
 - A Query with an unspecified filter defaults to a filter of "true", rather than "false".

- A Query with an unspecified candidate Extent but a specified candidate Class will automatically create an Extent of the appropriate type with subclasses turned on.
- Various bugs related to compiled queries with null arguments or no parameters have been resolved.
- Various InstanceCallbacks interface bugs have been resolved.
- Ant schematool and jdoc tasks deal with the Ant ClassLoader system better.
- Notable changes
 - Made `GenericDictionary.toSQL()` and `GenericDictionary.fromSQL()` methods `final`. Subclasses of `GenericDictionary` that must change the behavior of SQL generation or parsing should do so by overriding the appropriate `xxxToSQL()` or `xxxFromSQL()` methods instead. Note that the source for all our dictionary classes is now available in the Kodo JDO distribution.

2.3.0

- New features
 - JDO specification version 1.0 support.
 - Highly flexible multiple-table inheritance model now supported. See the multi-table class mapping documentation for details.
 - Support for large result sets when using any JDBC 2.0+ driver that supports ResultSets of type `TYPE_SCROLL_INSENSITIVE`. Return values from all `Query.executeXXX()` methods will be an instance of `java.util.List`, which can then be used for efficient random access.
 - DataCache API batches distributed updates, facilitating custom processing of distributed cache invalidation.
 - DataCache implementation loads data read from the data store into the cache as well as data being written to the data store.
 - JDBC back-end customizability is improved, allowing for custom field and class mappings and much finer-grained control of generated SQL.
 - Kodo JDO now supports extending JDOQL with custom tags. A number of default extensions, including substring searches and case-insensitive searches, are included by default with Kodo JDO. For more on this feature, see the query extensions documentation.
 - Support for IBM WebSphere, and other application servers that do not provide a TransactionManager through a JNDI lookup.
 - Source code release for various utility classes under the `source/` directory.
 - Deprecated the `srcDir` attribute of `jdoc` and `schematool`: the nested fileset no longer needs to be relative to an absolute directory.
 - Added a new method to `ExtentImpl` that returns a list containing all objects described by the extent.
- Bugfixes
 - Resolved a problem with large (> 5000 bytes) BLOBs being stored in Oracle.

- Fixed problem with compiled queries and null parameters returning empty result sets. Currently, when null parameters are used, prepared statements are bypassed and custom SQL is generated instead. In a future release, a new prepared statement that uses IS NULL will be generated on-the-fly.

2.2.5 May 6, 2002

- New features
 - The String serialization of ObjectIds.Id now uses a '-' as the delimiter, as the previous choice of the '#' made it difficult to use the serialization in a JSP without re-encoding it.
 - Released ant tasks for JDO enhancement, the SchemaTool, and an XDoclet task for generating .jdo metadata files from java source code comments.
 - Integration features for the upcoming JBuilder 7.
- Bugfixes
 - Fixed issue with managed transaction rollbacks. See bug #157 for details.
 - Fixed problem with prepared statements and in-memory queries. See bug #161 for details.

2.2.4 SP1 April 19, 2002

- Bugfixes
 - Resolved issues when using null parameters and compiled queries.

2.2.4 April 17, 2002

- New features
 - Support for java.math.BigInteger and java.math.BigDecimal
 - Added support for using packagename.jdo as the package's JDO metadata file, where "packagename" is the last section of the resource's package. E.g., a java class named com.solarmetric.mypackage.MyClass can now use a metadata file named mypackage.jdo.
 - Query.compile() will now create and use a PreparedStatement.
 - Kodo JDO now supports both single-JVM and distributed caching of persistent data.
 - It is now possible to extend the PersistenceManagerImpl and the EEPersistenceManager.
 - Added support for serialization/deserialization of an object id to/from a String.
 - Added an example of using Kodo within JSPs in the samples/jsp/ directory.
- Bugfixes

- Resolved inefficient behavior when executing a query that returns objects that have already been loaded but are hollow (bug 116).
- Fixed `NotSerializableException` when trying to bind an instance of `EEPersistenceManagerFactory` into JNDI (bug 117).
- Fixed problem where changing any of the configuration values in a `PersistenceManagerFactory` changed those values for all `PersistenceManagerFactory` instances on the system (bug 131).

2.2.3 March 4, 2002

- New features
 - New and improved documentation is now available at docs/manual.html. Enjoy!
 - Added code to check parameter count against declared parameter count when executing queries.
 - Partial support for the Java Connector Architecture is now available in the Enterprise Edition. This permits simple configuration of Kodo JDO using an application server's JCA configuration tools.
 - `PersistenceManagerFactory` instances can now be created from a `Properties`.
 - Guaranteed that SQL statements corresponding to object modifications (insert, update, delete) occur in the order that the modifications were performed in the `PersistenceManager`. If an object is modified multiple times, it will remain in the position that it was in after its first modification. When committing, we now traverse this list in order, so it is possible to do things like delete an object and then add a new object with the same id in a single transaction.
 - Added optional '-outfile filename' option to `schematool`. If specified, the SQL statements necessary to perform the schema modification will be appended to filename. No changes will be made to the database itself. This is useful when database modification is not permitted, or for post-processing the SQL generated by Kodo JDO with an external tool.
 - Changed `schematool` to print a warning when an array, collection, or map field is implicitly made persistent because of the rules of the spec. This often leads to undesirable behavior, as the default mode of insertion is to serialize the array/collection/map into a BLOB field, which is more often than not the desired behavior.
 - Both 'kodo' and 'tt' are now supported vendor tags. No collision checking is performed, so you should probably use just one.
 - Added support for Hypersonic free file-based JDBC driver
 - Added a new database preference: `db/schema-name`. If set, this value will be used in calls to `DatabaseMetaData.getTables()`.
- Bugfixes
 - Made queries take into account changes in the current transaction if `IgnoreCache == false`.
 - Made extents pay attention to changes made in the current transaction
 - Changed methods that are part of `javax.jdo` interfaces to never throw anything but `JDOExceptions`. See bug 69.
 - Resolved problem with listing table names when multiple database users should each have their own set of

tables. See bug 77.

- Only invalidate the connection and not return it to the pool if the Connection name contains "postgres". See PostgreSQL bugfix in 2.2.2 section for more details.
- Fixed a problem where executing 'jdoc' on a package.jdo that contains both app id and datastore id classes causes a failure.
- Improved error messages.

2.2.2 February 14, 2002

- New features
 - Added a duplicate column check to SQL INSERT and UPDATE query generation methods. If a duplicate column name is encountered and the values are also duplicates, then life proceeds happily along. If duplicates are found and the values differ, a JDOUserException is thrown. This permits using schema mappings in which a column is used both as a primary key and a foreign key.
- Bugfixes
 - Resolved potential deadlocks. See bug 42.
 - Added mechanism for controlling date precision when constructing SQL statements. See bug 6.
 - Fixed schematool strangeness when using table name metadata extensions. See bug 54.
 - improved error-reporting in exceptions thrown when invalid data is added to proxy collections/maps.
 - Fixed bug in which persistent-deleted objects were not containing the correct values on rollback if RetainValues was set. This fix makes persistent-deleted objects transition to hollow instead of performing any rollback.
 - Transaction.commit() and Transaction.rollback() now throw a JDOUserException instead of an IllegalStateException when a transaction is not active. See bug 44.
 - Because of a probable Postgres JDBC driver bug, changed connection pooling to not recycle connections that have been involved in a transaction.
 - Resolved a VerifyError that occurred when a non-primitive, non-String object was used as part of an object's primary key.
 - Resolved a situation in which the number of connections needed to load a single object from the data store was proportional to the depth of persistence-capable fields in the tree of default fetch groups. That is, if A has a relation to B called b, and B has a relation to C called c, and b is in A's default fetch group, and c is in B's default fetch group, then three connections were needed in order to load an A.
 - Fixed bug in which queries on date fields occasionally threw exceptions.
 - Fixed obscure bug in makeDirty(). If using data store transactions and setting a JDO field without first having loaded the field (either implicitly by having the field in the default fetch group, or explicitly), then the field would not be set when InstanceCallbacks.jdoPostLoad() was invoked. Additionally, nontransactionalRead must have been set to true for this problem to occur.
 - Fixed a bug that caused queries to fail in certain Tomcat configurations. See bug 35.

- Added `writeReplace()` methods to fix issues with serialization of dates and collection types retrieved from data stores.
- Fixed bug in which `jdoNewInstance(StateManager,Object)` method was only being added to base application identity classes.
- `com.solarmetric.kodo/runtime/PersistenceManagerImpl.java`: improved error reporting when validating and making persistent objects that are not managed by the current PM.
- Notable changes
 - Made default table type for MySQL be Berkley DB, which has real transactional capabilities
 - Set the default to warn on persistent type failures, rather than throw an exception.

2.2.1 November 1 2001

- New features
 - IBM DB2 UDB 7.2 is now supported.
 - All datastore identity classes now use the `com.solarmetric.kodo.util.ObjectIds.Id` object ID type rather than individual dynamically loaded classes.
 - Performance enhancements.
- Bugfixes
 - Old versions of MySQL for Windows are now supported.
 - A new algorithm for auto-generation of table/column/index names that is much less likely to generate naming collisions is now available.
 - Fixed `PersistenceManager.refreshAll()` behavior when no transaction is active.
 - New persistent objects, first class children, etc. are correctly dealt with when created in `jdoPreStore()`.
 - Queries that perform multiple `contains()`, `containsKey()`, or `containsValue()` clauses &&'d together for different values on the same collection/map now work.
 - The PM will throw some subclass of `JDOFatalException` on commit if and only if the transaction is also automatically rolled back.
- Notable changes
 - One-to-one mappings are dealt with more efficiently, reducing the number of database accesses and therefore improving performance.
 - Changed resource-loading and class-loading to use the current thread's context's class loader, rather than the system class loader. This makes deployment to a web application container much easier.
 - A single class is now used as the ID class for all persistent types managed with data store identity.

2.2.0 October 5, 2001

- New features
 - Application identity is now supported.
 - Preview release of tool to generate a class suitable for use as an application-identity object id class, complete with an appropriate equals() method, a corresponding hashCode() method, and a toString() method. For more information and usage, run 'java com.solarmetric.kodo.tools.appid.ApplicationIDTool'.
 - Improvements have been made to common error messages, and inappropriate exception types have been replaced with more useful ones.
 - New library: kodo-jdo-runtime.jar. This library contains all the class files necessary for run-time use of Kodo JDO.
 - Enhanced mapping customizations for mapping application-identity pk fields (see docs/existing-schema.html)
 - Various minor performance enhancements
- Bugfixes
 - PersistenceManager refresh methods behave correctly when invoked from outside the context of a transaction. Note that the noargs refreshAll () call behaves as designated by the JDO javadoc, not as designated by the 0.95 specification.
 - SQL generation for statements that insert decimals (floats and doubles) now always use United States notation (3.14159 for example).
 - Assorted minor bugfixes.
- Notable changes
 - Major redesign of the refresh mechanism.

beta 2.1 July 15, 2001

- New features
 - The null-value attribute on field metadata is supported.
 - BLOB mappings are supported; any serializable field can now be persisted. (Note: PostgreSQL does not support BLOB mappings)
- Bugfixes
 - jdoPreStore() is no longer called on deleted instances.
 - Fixed a NullPointerException that could occur when softly-cached instances were garbage collected by the JVM.

- Indexes were not being created on fields marked with the 'column-index' metadata extension.
- Fixed a bug that prevented retrieving CLOB values with Oracle.
- Fixed a bug that caused a SQLException with fields set to empty strings or chars with a 0 value on PostgreSQL.
- Notable changes
 - The SQLTypeMap, used in DBDictionaries, changed slightly.
 - The Kodo User Guide chapter on Metadata has been updated to include information on the new 'blob' metadata extension for explicitly marking fields that should be stored in serialized form.

beta 2 July 10, 2001

- New features
 - Maps with user-defined persistent object values can be persisted (n-many relations).
 - Static inner classes can be persisted.
 - Queries support the use of containsKey() and containsValue() for Map fields.
 - Queries support ordering declarations.
 - The SchemaTool's schema migration capabilities have been enhanced.
 - The SchemaTool offers the option of automatically maintaining the list of persistent types for the system.
 - Schema generation can be customized through JDO metadata.
 - The standard javax.sql.DataSource is used to obtain connections.
 - Connection pooling has been enhanced, and new pooling parameters have been added (timeout time, autoreturn time).
 - The ObjectId helper class has been introduced to map opaque JDO OID values to and from primitive long values.
 - PersistenceManagerFactories can be stored in JNDI, including JNDI trees that are replicated over multiple JVMs.
 - PersistenceManagers can transparently synchronize with global J2EE Transactions (Kodo Enterprise Edition beta only).
- Bugfixes
 - Row-level locking is now performed within pessimistic Transactions.
 - Object ID generation is now done using the database by default.
 - Globally unique primary key values are no longer required (per-class only).
 - Inserting new instances of classes mapped to an existing schema without a class indicator column no longer

throws a `NullPointerException`.

- Numerous minor fixes.
- Notable changes
 - Users of previous beta versions of Kodo should scan the user guide in the `docs/` directory for new information included with this release.
 - The schematool can now automatically maintain a list of persistent classes; the `persistent-types` array in `system.prefs` is not needed. This is covered in the Database Setup chapter of the user guide.
 - The syntax for using the schematool has changed. This is covered in the Database Setup chapter of the user guide.
 - The syntax for mapping classes to existing database tables has changed. This is covered in the Using Existing Schema chapter of the user guide.
 - The Runtime Use chapter of the user guide covers new runtime options available, such as JNDI storage of the `JDBCPersistenceManagerFactory` and safe conversion of JDO OID values to and from primitive long values.

Appendix F. Known Bugs and Limitations

The following represents a list of significant known bugs and limitations of Kodo JDO. These features were not ready in time for this release, but are expected to be added to future releases:

- Non-static inner classes cannot be persisted.
- Static fields cannot be persisted.
- Fields of an unknown type (i.e. `java.lang.Object`) or a user-defined type that is not persistence-capable must be serializable.
- Fields of type `Locale` are persisted through serialization only.
- Persistent fields of any `Map` type must use simple key classes (i.e. `String`, `Integer`, `Boolean`, etc) or will be persisted through serialization only.
- Queries do not support the `~` operator.
- Queries do not support the `-` operator used for unary negation (it can be used for subtraction and to represent negative numbers).
- If a query ordering that traverses a relation is used, but that relation is not traversed in the filter, then no join will be performed and the resultant collection will therefore be incorrect.
- Queries do not support declared variables that are not bound in a `contains()`, `containsKey()`, or `containsValue()` clause.
- Fields persisted through serialization cannot be used in queries.
- Using a cast in a Query made against an Extent may not prevent instances of the base class from being returned.
- The SchemaTool does not detect secondary tables that are no longer used.
- Kodo JDO Enterprise Edition does not support redeployment of EJBs.
- Queries that use parameters for matching fields whose underlying data store type is CLOB (string fields with `column-length` set to `-1`) cannot be compiled.
- Queries that use parameters as values in `startsWith()` or `endsWith()` calls must manually concatenate a `%` to the parameter passed to `execute()`

Additionally, the following database and JVM specific limitations exist:

Sun JDK 1.2.2 on Solaris

- There seem to be bytecode-level incompatibilities between Kodo and Sun's 1.2.2 JDKs for Solaris. Sun's 1.3 JDKs for Solaris, however, work without problems. This issue is being investigated.

InstantDB

- Pessimistic locking is not supported.
- Using `isEmpty()` in queries made against an Extent will fail because SQL sub-selects are not supported.

MySQL

- Using `isEmpty()` in queries made against an Extent will fail because SQL sub-selects are not supported.
- Rollback due to database error or optimistic lock violation is not supported unless the table type is one of the MySQL transactional types. Explicit calls to `rollback()` before a transaction has been committed, however, are always supported.
- Floats and doubles may lose precision when stored in some data stores.
- When storing a field of type `java.math.BigDecimal`, some data stores will add extraneous trailing 0 characters, causing an equality mismatch between the field that is stored and the field that is retrieved.

PostgreSQL

- Pessimistic locking is not supported on queries that use `SELECT DISTINCT` (i.e. any query that uses `contains()`, `containsValue()`, or `containsKey()`). Modifying objects found using such queries in pessimistic transactions is permitted but may result in an optimistic lock exception if the same instances are also modified by another concurrent thread.
- Floats and doubles may lose precision when stored.
- PostgreSQL cannot store very low and very high Dates.
- Empty string/char values are stored as NULL.
- BLOBs are not supported; fields cannot be stored via serialization.

IBM DB2

- Floats and doubles may lose precision when stored.
- Empty char values are stored as NULL.
- Pessimistic locking is not supported.
- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending `DB2Dictionary`

Oracle

- Pessimistic locking is not supported on queries that use `SELECT DISTINCT` (i.e. any query that uses `contains()`, `containsValue()`, or `containsKey()`). Modifying objects found using such queries in pessimistic transactions is permitted but may result in an optimistic lock exception if the same instances are also modified by another concurrent thread.
- Floats and doubles may lose precision when stored.
- CLOB columns cannot be used in queries.

SQLServer

- Floats and doubles may lose precision when stored.
- TEXT columns cannot be used in queries.

Sybase

- Datastore locking cannot be used when manipulating many-to-many relations using the default Kodo schema created by the schematool, unless an auto-increment primary key field is manually added to the table.

PointBase

- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending `PointbaseDictionary`

Please see the Kodo JDO bug tracking database for an up-to-date bug list.

Appendix G. Supported Databases

Following is a table of the database versions and JDBC driver versions that are supported by Kodo JDO.

Table G.1. Supported Databases and JDBC Drivers

| Database Name | Database Version | JDBC Driver Name | JDBC Driver Version |
|-----------------------------------|----------------------------|--------------------------|---------------------|
| Pointbase | 4.2 | Pointbase JDBC driver | 4.2 (4.2) |
| InstantDB | 3.26 | InstantDB JDBC driver | 3.26 (3.26) |
| Cloudscape | 4.0.6 | Cloudscape JDBC driver | 4.0 (4.0.6) |
| DB2 | 7.2 | DB2 JDBC driver | 7.1 (7.1) |
| Oracle | 8.1-9.1 | Oracle JDBC driver | 9.0 (9.0.1.0.0) |
| PostgreSQL | 7.2.1 | PostgreSQL Native Driver | 7.2 (7.2) |
| Microsoft SQL Server | 8.00.194 (SQL Server 2000) | SQLServer | 2.2 (2.2.0002) |
| Sybase Adaptive Server Enterprise | 12.5 | jConnect | 4.2 (4.2) |
| Hypersonic Database Engine | 1.6 | Hypersonic | 1.6 (1.6) |
| MySQL | 3.23.43-log | MySQL Driver | 2.0 (2.0.14) |

G.1. Example properties for Pointbase JDBC driver (Pointbase)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: \
    jdbc:pointbase:jdotest,database.home=pointbasedb,create=true,cache.size=10000,database.pagesize=
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: \
    com.pointbase.jdbc.jdbcUniversalDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.2. Example properties for InstantDB JDBC driver (InstantDB)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:ldb:lib/ldb_sample.properties
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: \
    org.enhydra.instantdb.jdbc.ldbDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.3. Example properties for Cloudscape JDBC driver (Cloudscape)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:cloudscape:COUDSCAPE_DB;create=true
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: COM.cloudscape.core.JDBCdriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.4. Example properties for DB2 JDBC driver (DB2)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:db2://uranus.solarmetric.com/jdotest
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: COM.ibm.db2.jdbc.net.DB2Driver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.5. Example properties for Oracle JDBC driver (Oracle)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:oracle:thin:@saturn:1521:solarsid
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: oracle.jdbc.driver.OracleDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.6. Example properties for PostgreSQL Native Driver (PostgreSQL)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:postgresql://saturn:5432/jdotest
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: org.postgresql.Driver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.7. Example properties for SQLServer (Microsoft SQL Server)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotest
javax.jdo.option.ConnectionURL: \
    jdbc:microsoft:sqlserver://uranus:1433;DatabaseName=jdotest;SelectMethod=cursor
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: \
    com.microsoft.jdbc.sqlserver.SQLServerDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.8. Example properties for jConnect (Sybase Adaptive Server Enterprise)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotest
javax.jdo.option.ConnectionURL: \
    jdbc:sybase:Tds:saturn:4100/jdotest?ServiceName=jdotest&BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: com.sybase.jdbc.SybDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.9. Example properties for Hypersonic (Hypersonic Database Engine)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: sa
javax.jdo.option.ConnectionURL: jdbc:hsqldb:hsqldb_sample_database
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10

```

G.10. Example properties for MySQL Driver (MySQL)

```

javax.jdo.option.RetainValues: true
javax.jdo.option.RestoreValues: true

```

```
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.MsWait: 5000
javax.jdo.option.MinPool: 1
javax.jdo.option.MaxPool: 80
javax.jdo.option.IgnoreCache: false
javax.jdo.option.ConnectionUserName: jdotestuser
javax.jdo.option.ConnectionURL: jdbc:mysql://saturn/jdotest
javax.jdo.option.ConnectionPassword: PASSWORD
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure: true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass: \
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping: true
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
com.solarmetric.kodo.EnableQueryExtensions: false
com.solarmetric.kodo.DefaultFetchThreshold: 30
com.solarmetric.kodo.DefaultFetchBatchSize: 10
```