

SolarMetric Kodo JDO 2.5.8 Developers Guide

SolarMetric Kodo™ JDO 2.5.8 Developers Guide

Copyright © 2002, 2003 SolarMetric Inc.

Table of Contents

I. Introduction	1
1. Introduction to SolarMetric Kodo JDO	2
II. Java Data Objects	3
1. Introduction	4
1.1. Intended Audience	4
1.2. Transparent Persistence	4
2. Why JDO?	5
3. JDO Architecture	7
3.1. JDO Exceptions	7
4. PersistenceCapable	8
4.1. Enhancers	8
4.2. Persistence-Capable vs. Persistence-Aware	9
4.3. Restrictions on Persistent Classes	9
4.3.1. Inheritance	10
4.3.2. Persistent Fields	10
4.3.3. Conclusions	12
4.4. InstanceCallbacks	12
4.5. JDO Identity	13
4.5.1. Datastore Identity	14
4.5.2. Application Identity	14
4.6. Conclusions	15
5. Metadata	16
5.1. Metadata DTD	16
5.2. Metadata Placement	20
6. JDOHelper	22
6.1. Persistence-Capable Operations	22
6.2. Lifecycle Operations	22
6.3. PersistenceManagerFactory Construction	25
7. PersistenceManagerFactory	26
7.1. Obtaining a PersistenceManagerFactory	26
7.2. PersistenceManagerFactory Properties	27
7.2.1. Connection Configuration	27
7.2.2. PersistenceManagerFactory and Transaction Defaults	27
7.2.3. PersistenceManager Pooling	28
7.3. Obtaining PersistenceManagers	29
7.4. Properties and Supported Options	29
8. PersistenceManager	31
8.1. User Object Association	32
8.2. Configuration Properties	32
8.3. Transaction Association	32
8.4. Persistence-capable Lifecycle Management	32
8.5. JDO Identity Management	33
8.6. Extent Factory	34
8.7. Query Factory	34
8.8. Closing	34
9. Transaction	35
9.1. Transaction Types	35
9.2. The JDO Transaction Interface	36
10. Extent	38
11. Query	39
11.1. Required Query Elements	39
11.2. Optional Query Elements	39
11.3. JDOQL	40

11.4. Executing Queries	43
11.5. Query Compilation	43
III. Kodo JDO Reference Guide	44
1. Introduction	45
1.1. Intended Audience	45
2. Configuration Framework	46
2.1. JDO Standard Properties	46
2.1.1. javax.jdo.PersistenceManagerFactoryClass	47
2.1.2. javax.jdo.option.Optimistic	47
2.1.3. javax.jdo.option.Multithreaded	47
2.1.4. javax.jdo.option.IgnoreCache	47
2.1.5. javax.jdo.option.RetainValues	48
2.1.6. javax.jdo.option.RestoreValues	48
2.1.7. javax.jdo.option.NontransactionalRead	48
2.1.8. javax.jdo.option.NontransactionalWrite	48
2.1.9. javax.jdo.option.ConnectionURL	48
2.1.10. javax.jdo.option.ConnectionUserName	49
2.1.11. javax.jdo.option.ConnectionPassword	49
2.1.12. javax.jdo.option.ConnectionDriverName	49
2.1.13. javax.jdo.option.ConnectionFactoryName	49
2.1.14. javax.jdo.option.ConnectionFactory2Name	49
2.1.15. javax.jdo.option.MinPool	50
2.1.16. javax.jdo.option.MaxPool	50
2.1.17. javax.jdo.option.MsWait	50
2.2. Kodo JDO Properties	50
2.2.1. com.solarmetric.kodo.CacheReferenceSize	50
2.2.2. com.solarmetric.kodo.ConnectionProperties	51
2.2.3. com.solarmetric.kodo.DataCacheClass	51
2.2.4. com.solarmetric.kodo.DataCacheProperties	52
2.2.5. com.solarmetric.kodo.RemoteCommitProviderClass	52
2.2.6. com.solarmetric.kodo.RemoteCommitProviderProperties	52
2.2.7. com.solarmetric.kodo.DefaultDataCacheTimeout	52
2.2.8. com.solarmetric.kodo.DefaultFetchThreshold	53
2.2.9. com.solarmetric.kodo.DefaultFetchBatchSize	53
2.2.10. com.solarmetric.kodo.EnableQueryExtensions	53
2.2.11. com.solarmetric.kodo.FetchGroups	53
2.2.12. com.solarmetric.kodo.FlushBeforeQueries	53
2.2.13. com.solarmetric.kodo.LicenseKey	54
2.2.14. com.solarmetric.kodo.PersistenceManagerClass	55
2.2.15. com.solarmetric.kodo.PersistenceManagerProperties	55
2.2.16. com.solarmetric.kodo.ProxyManagerClass	55
2.2.17. com.solarmetric.kodo.ProxyManagerProperties	55
2.2.18. com.solarmetric.kodo.QueryCacheClass	55
2.2.19. com.solarmetric.kodo.QueryCacheProperties	56
2.2.20. com.solarmetric.kodo.QueryFilterListeners	56
2.2.21. com.solarmetric.kodo.ResultListClass	56
2.2.22. com.solarmetric.kodo.ResultListProperties	57
2.2.23. com.solarmetric.kodo.TransactionCacheClass	57
2.2.24. com.solarmetric.kodo.TransactionCacheProperties	57
2.2.25. com.solarmetric.kodo.UseSoftTransactionCache	58
2.2.26. com.solarmetric.kodo.PersistentTypes	58
2.2.27. com.solarmetric.kodo.TransactionMode	58
2.2.28. com.solarmetric.kodo.ConnectionFactory2Properties	58
2.2.29. com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode	59
2.2.30. com.solarmetric.kodo.impl.jdbc.ConnectionTestTimeout	59
2.2.31. com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass	60
2.2.32. com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass	60
2.2.33. com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderProperties	60

2.2.34. com.solarmetric.kodo.impl.jdbc.DictionaryClass	60
2.2.35. com.solarmetric.kodo.impl.jdbc.DictionaryProperties	61
2.2.36. com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping	63
2.2.37. com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass	63
2.2.38. com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties	64
2.2.39. com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass	65
2.2.40. com.solarmetric.kodo.impl.jdbc.SQLExecutionListenerClass	65
2.2.41. com.solarmetric.kodo.impl.jdbc.StatementCacheMaxSize	65
2.2.42. com.solarmetric.kodo.impl.jdbc.StatementExecutionTimeout	65
2.2.43. com.solarmetric.kodo.impl.jdbc.SynchronizeSchema	66
2.2.44. com.solarmetric.kodo.impl.jdbc.TransactionIsolation	66
2.2.45. com.solarmetric.kodo.impl.jdbc.UseBatchedStatements	66
2.2.46. com.solarmetric.kodo.impl.jdbc.UseSQL92Joins	67
2.2.47. com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure	67
2.2.48. com.solarmetric.kodo.ee.ManagedRuntimeClass	67
2.2.49. com.solarmetric.kodo.ee.ManagedRuntimeProperties	67
2.3. Logging Framework	68
2.3.1. Disabling Logging	69
2.3.2. Logging using Apache Log4J	69
2.3.3. Logging using JDK 1.4 java.util.logging	70
2.3.4. Logging using Simple Log	71
2.3.5. Logging using a Custom Log	71
3. Creating Persistent Classes	73
3.1. Application Identity Class Generation	73
3.2. Enhancement	73
3.3. Auto-Generating Classes from a Schema	74
3.3.1. Customizing Reverse Mapping	77
3.3.2. Formatting Reverse Mapping Code	78
3.3.3. Schema File DTD	78
3.4. Smart Proxies	79
4. Metadata	80
4.1. Dependency Extensions	80
4.1.1. dependent	80
4.1.2. element-dependent	80
4.1.3. value-dependent	80
4.1.4. key-dependent	80
4.2. Class-level Object-Relational Mapping Extensions	80
4.2.1. table	80
4.2.2. pk-column	81
4.2.3. lock-column	81
4.2.4. class-column	81
4.2.5. subclass-provider	81
4.2.6. subclass-indicator-value	81
4.2.7. custom-mapping	81
4.2.8. can-cache	81
4.2.9. data-cache-name	82
4.2.10. cache-timeout	82
4.2.11. sequence-factory-class	82
4.2.12. sequence	82
4.2.13. sequence-factory-properties	82
4.3. Field-level Object-Relational Mapping Extensions	82
4.3.1. dependent	82
4.3.2. element-dependent	82
4.3.3. value-dependent	82
4.3.4. key-dependent	82
4.3.5. inverse	83
4.3.6. blob	83
4.3.7. column-length	83

4.3.8. column-index	83
4.3.9. ordered	83
4.3.10. table	83
4.3.11. data-column	83
4.3.12. key-column	84
4.3.13. ref-column	84
4.3.14. order-column	84
4.3.15. read-only	84
4.3.16. <tablename>-pk-column	84
4.3.17. fetch-group	84
4.3.18. custom-mapping	84
4.3.19. externalizer	84
4.3.20. factory	84
4.4. Extensions Under Application Identity	85
4.5. Examples	85
4.5.1. Mapping Classes to an Existing Schema	85
4.5.2. Mapping One to Many relations	87
4.5.3. Extensions Under Application Identity	88
4.6. Multi-table Inheritance Mapping	90
4.7. Generating Default JDO Metadata	90
5. JDBC Configuration	92
5.1. Supported Databases	92
5.2. Accessing Multiple Databases	92
5.3. Connection Management	92
5.4. Large Result Sets	93
5.5. Schema Manipulation	94
6. Standard Features	96
6.1. Manipulating Datastore Identity Objects	96
6.2. ExtentImpl	96
6.3. PersistenceManager Extension	96
6.4. Event Notification Framework	96
6.4.1. Transaction Event Notification	96
6.4.2. Remote Commit Notification	97
6.5. Custom Proxies	97
6.6. Access to SQL Connections	97
6.7. PersistenceManagerImpl.evictAll() API extensions	98
6.8. Custom Class Indicators	98
6.9. Storing Second Class Objects via Stringification	99
6.10. Primary Key Generation	101
6.10.1. Using table-based key generation	101
6.10.2. Using Database Sequences for key generation	102
6.11. StateManagerSet configuration	102
7. Enterprise Features	104
7.1. Datastore Cache	104
7.1.1. Overview of Kodo JDO Datastore Caching	104
7.1.2. Kodo JDO Cache Usage	105
7.1.3. Kodo JDO Query Caching	106
7.1.4. Kodo JDO Data Cache Configuration	108
7.1.5. Cache Extension	109
7.1.6. Important notes about the DataCache	109
7.1.7. Known issues and limitations	109
7.2. Query Extensions	110
7.2.1. Using Query Extensions	110
7.2.2. Included Query Extensions	111
7.2.3. Deprecated Query Extensions	112
7.2.4. Developing Custom Query Extensions	112
7.2.5. Configuring Query Extensions	112
7.3. Fetch Groups	112

7.3.1. Normal Default Fetch Group Behavior	113
7.3.2. Kodo JDO Fetch Group Behavior	114
7.3.3. Configuring a PersistenceManager to load Fetch Groups	115
7.4. XA Transactions	115
7.4.1. Overview of XA Distributed Transaction Processing	115
7.4.2. Requirements for using Kodo with XA transactions	115
7.4.3. Configuring Kodo to utilize XA transactions	115
7.5. Remote Commit Notification Framework	116
7.5.1. Kodo JDO RemoteCommitProvider Configuration	116
7.5.2. Event Notification Framework Customization	118
8. Additional Features	119
8.1. Custom data processing	119
8.2. Custom data requests	119
9. Third Party Integration Features	121
9.1. Overview of Third Party Integration features in Kodo	121
9.2. Apache Ant	121
9.2.1. Common Ant Configuration Options	121
9.2.2. JDOEnhancer Ant Task	122
9.2.3. SchemaTool Ant Task	123
9.3. XDoclet	123
9.4. Borland JBuilder	126
9.4.1. Installing Kodo into JBuilder	127
9.4.2. Kodo Configuration from JBuilder	127
9.4.3. Creating and building JDO projects in JBuilder	127
9.4.4. Editing JDO Metadata from JBuilder	128
9.4.5. Running the SchemaTool from JBuilder	128
9.4.6. JBuilder Project Sample	128
9.5. Sun ONE Studio / NetBeans IDE	128
9.5.1. Before Installing Kodo into the IDE	128
9.5.2. Installing Kodo into the IDE	129
9.5.3. Configuring the Kodo Module	129
9.5.4. Kodo Template Wizards	129
9.5.5. JDO DataObject	130
9.5.6. Kodo Integration into the Build Process	130
9.5.7. SunONE / NetBeans Sample	130
9.6. Eclipse / WebSphere Studio Integration	130
9.6.1. Installing the Kodo Eclipse Plugin	131
9.6.2. Configuring the Plugin	131
9.6.3. Using Kodo in Eclipse IDEs	131
9.6.4. Eclipse Sample	132
10. Enterprise Integration	133
10.1. Overview of Enterprise Integration features in Kodo	133
10.2. Using Kodo JDO via the Java Connector Architecture	134
10.2.1. Overview of the JCA	134
10.2.2. Deploying on JBoss 3.0	134
11. Kodo JDO Implementation Notes	135
11.1. PersistenceCapable.jdoFlags field and fields in the Default Fetch Group	135
11.2. Optimistic locking mechanism	135
12. Optimization Techniques	136
IV. Kodo JDO Tutorial	140
1. Introduction to the Kodo JDO Tutorial	cxli
1.1. Tutorial Requirements	cxli
1. The Pet Shop	142
1.1. Included Files	142
1.2. Important Utilities	143
2. Getting Started	144
3. Inventory Maintenance	146
4. Inventory Growth	149

5. Behavioral Analysis	151
6. Extra Features	156
V. Kodo J2EE Tutorial	157
1. Introduction to the J2EE Tutorial	clviii
1.1. Prerequisites for the Kodo J2EE Tutorial	clviii
1. J2EE Installation Types	159
2. Installing Kodo JCA	160
2.1. JBoss 3.0	160
2.2. JBoss 3.2	160
2.3. WebLogic Versions 6.2 to 7.x	160
2.4. WebLogic 8.1	160
2.5. WebSphere 5	161
2.6. SunONE Application Server	162
2.7. Borland Enterprise Server 5.2	162
3. Installing the J2EE Sample Application	164
3.1. Compiling and Building The Sample Application	164
3.2. Deploying Sample To JBoss	164
3.3. Deploying Sample To WebLogic 6.2 to 7.x	164
3.4. Deploying Sample To WebLogic 8.1	165
3.5. Deploying Sample To SunONE	165
3.6. Deploying Sample To WebSphere	165
3.7. Deploying Sample To Borland Enterprise Server 5.2	165
4. Using The Sample Application	166
5. Sample Architecture	167
6. Code Notes and J2EE Tips	168
VI. Reverse Mapping Tool Tutorial	170
1. Introduction to the Reverse Mapping Tool Tutorial	clxxi
1.1. Reverse Mapping Tool Tutorial Requirements	clxxi
1. Magazine Shop	172
2. Setup	173
2.1. Tutorial Files	173
2.2. Important Utilities	173
3. Generating Persistent Classes	174
4. Using the Finder	176
A. Development and Runtime Libraries	177
B. JDO Resources	178
C. Optional JDO Features	179
D. SQL Types	180
E. Release Notes	182
F. Known Bugs and Limitations	203
G. Supported Databases	206
G.1. Example properties for Pointbase	206
G.2. Example properties for IBM DB2	206
G.3. Example properties for Informix Dynamic Server	206
G.4. Example properties for Oracle	207
G.5. Example properties for PostgreSQL	207
G.6. Example properties for SQLServer	207
G.7. Example properties for Sybase	207
G.8. Example properties for Hypersonic	207
G.9. Example properties for MySQL	207
H. Common Database Errors	209

List of Tables

2.1. Kodo Automatic Flush Behavior	54
7.1. Data access methods	104
D.1. SQL Type Mappings	180
D.2. SQL Type Mappings II	180
D.3. SQL Type Mappings III	180
G.1. Supported Databases and JDBC Drivers	206
H.1. Known Database Error Codes	209

List of Examples

4.1. PersistenceCapable Class	8
4.2. Accessing Mutable Persistent Fields	11
5.1. Basic Structure of Metadata Documents	16
5.2. Metadata Class Listings	18
5.3. Complete Metadata Document	20
6.1. Obtaining a PersistenceManagerFactory	25
9.1. Grouping Operations with Transactions	36
11.1. Basic Query	41
11.2. Result Ordering and Method Calls	41
11.3. Mathematical Operations and Relation Traversal	42
11.4. Precedence and Logical Operators	42
11.5. Imports and Parameters	42
11.6. Collections	42
11.7. Variables	43
2.1. Specifying Connection Properties	51
2.2. Specifying DataSource Properties	51
2.3. Specifying ConnectionFactory2 Properties	59
2.4. Example log4j.properties file for moderately verbose logging	69
2.5. Example log4j.properties file for disabled logging	70
2.6. Example log4j.properties file for debugging logging	70
2.7. Example logging.properties file	70
2.8. Example simplelog.properties file	71
2.9. Example custom logging class	71
3.1. Using the Application Identity Tool	73
3.2. Using the Kodo JDO Enhancer	74
3.3. Using the Reverse Mapping Tool	76
3.4. Customizing Reverse Mapping with Properties	77
3.5. Code Formatting with the Reverse Mapping Tool	78
5.1. Configuring custom dictionary properties	92
5.2. Properties for using a custom DataSource	93
5.3. Using Random Access Query Results in a Portable Fashion	93
5.4. Using the Kodo JDO Schematool	95
6.1. Obtaining a java.sql.Connection object from the PersistenceManager	98
6.2. IntegerSubclassProvider example	99
6.3. Metadata for persisting field of type java.lang.Class	100
6.4. Metadata for persisting field of type java.net.URL	100
6.5. Metadata for persisting field of type java.net.InetAddress	100
6.6. Metadata for persisting field of type java.util.TimeZone	101
6.7. Metadata for persisting field of type java.io.File	101
6.8. Properties for configuring DBSequenceFactory	102
6.9. Properties for configuring ClassSequenceFactory	102
6.10. Metadata for configuring DBSequenceFactory	102
7.1. Pinning an object into the DataCache	105
7.2. Unpinning an object from the DataCache	105
7.3. Evicting an object from the DataCache	106
7.4. Specifying a non-default DataCache	106
7.5. Notifying the query cache of altered classes	107
7.6. Dropping or pinning query results in the cache	107
7.7. Temporarily disabling and enabling query caching	108
7.8. Disabling query caching via configuration properties	108
7.9. Configuring a PersistenceManagerFactory to use a Tangosol cache for distributed cache needs	109
7.10. Use queries instead of extents	110
7.11. Configuring a PersistenceManagerFactory to use a JMS remote commit provider mechanism	117

7.12. Configuring a PersistenceManagerFactory to use a TCP remote commit provider mechanism	117
9.1. Using the <config> tag in an ant build.xml file	121
9.2. Using the properties attribute of the <config> tag	122
9.3. Using the <classpath> tag in an ant build.xml file	122
9.4. Invoking the JDOEnhancer from an Ant build.xml file	122
9.5. Invoking the SchemaTool from an Ant build.xml file	123
9.6. Invoking the JDO Metadata generator from an Ant build.xml file	124
9.7. Source code comments for automatic JDO Metadata generation	124
10.1. Binding a PersistenceManagerFactory into JNDI via a WebLogic startup class	133
10.2. Looking up the PersistenceManagerFactory in JNDI	134
12.1. Explicitly closing resources	137

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.

setup and use of the Kodo JDO implementation for relational databases.

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.

severe limitations. JDO attempts to overcome these limitations.

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`.

components are:

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` and `IllegalArgumentException` 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: Even with all of these advantages,

- 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.

Bytecode enhancers overcome the difficulties associated with source enhancers: 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.

few important limitations:

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`

assign a new value to the field. JDO supports the following immutable types for persistent fields:

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`

following mutable field type:

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`

following:

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

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.

following criteria:

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.

requirements:

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.

primary purposes:

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.

other `extension` elements as children, and has three attributes:

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`.

guidelines:

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.

map elements define four attributes. They are:

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: Assuming you are using a standard Java class loader, these rules imply that for a class `Magazine`

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.

standard locations: 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`

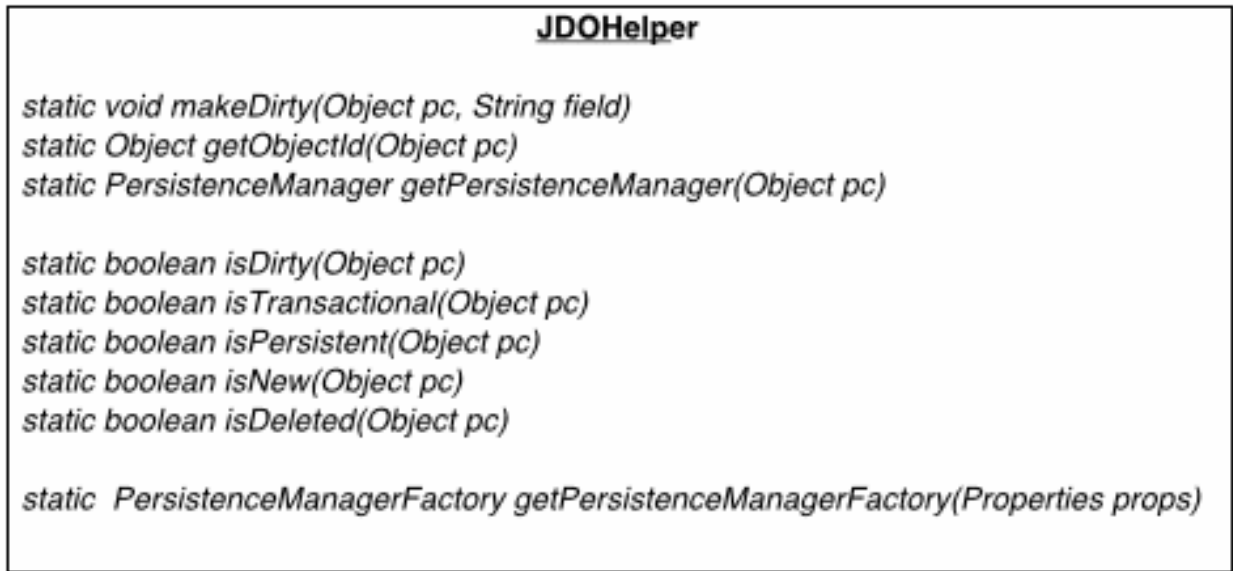
- `org/mag.jdo`

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.

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:hsqldb: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.

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

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. By default, Kodo automatically flushes changes that have occurred to the data store if this is set to `false` and Kodo detects that modifications in the current transaction may have an impact on the query being executed.

`JDOUnsupportedOptionException`.

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.

a minimum, the following keys:

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.NonDurableIdentity`, `javax.jdo.option.NullCollection` and `javax.jdo.option.ChangeApplicationIdentity`.

Chapter 8. PersistenceManager

PersistenceManager

Object UserObject
boolean Multithreaded
boolean IgnoreCache

Transaction currentTransaction()

void makePersistent(Object pc)
void makePersistentAll(...)
void deletePersistent(Object pc)
void deletePersistentAll(...)
void makeTransient(Object pc)
void makeTransientAll(...)
void makeTransactional(Object pc)
void makeTransactionalAll(...)
void makeNonTransactional(Object pc)
void makeNonTransactionalAll(...)
void evict(Object pc)
void evictAll(...)
void refresh(Object pc)
void refreshAll(...)
void retrieve(Object pc)
void retrieveAll(...)

Object getObjectById(Object oid, boolean validate)
Object getObjectById(Object pc)
Class getObjectByIdClass(Class pcClass)
Object newObjectByIdInstance(Class pcClass, String str)

Query newQuery(...)

Extent getExtent(Class pcClass, boolean includeSubclasses)

boolean isClosed()
void close()

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.

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

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: Together, these qualities are called the ACID

- *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.

in an all-or-nothing fashion. Transactions have the following qualities: 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.

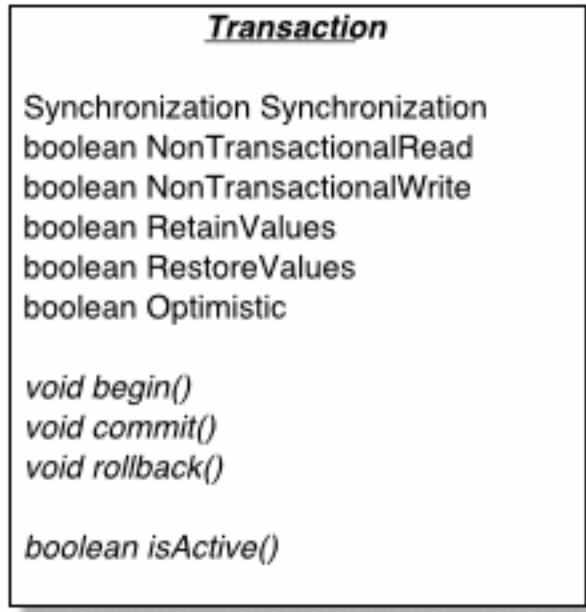
9.1. Transaction Types

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.

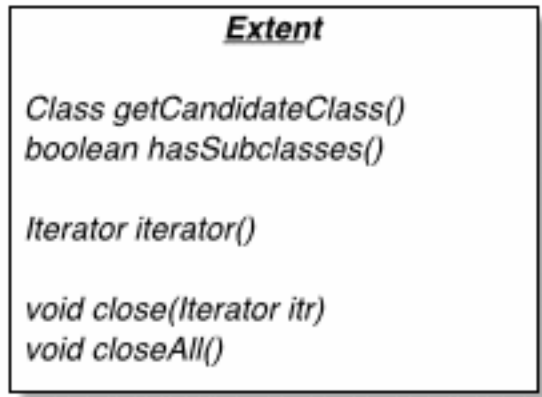
Finally, the `isActive` method returns true if the transaction is in progress (`begin` has been called more recently than `commit` or `rollback`), and false otherwise.

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

A rectangular box with a thin black border and a light gray drop shadow. Inside the box, the text is centered and formatted as follows:

```
Extent  
  
Class getCandidateClass()  
boolean hasSubclasses()  
  
Iterator iterator()  
  
void close(Iterator itr)  
void closeAll()
```

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.

Every *Query* has three required elements:

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.

The following are optional elements of JDO queries:

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`

Methods are not supported, with the following exceptions:

- 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.

as standard Java syntax, with the following exceptions:

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.

overrides defined by the `Configuration` interface:

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
```

(`String, int`). You could plug your cache into Kodo JDO by specifying the following properties:

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 connections that will not be enlisted in any global XA transaction.

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.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
```

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

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
```

depending on the `DataSource` in use: see the documentation for your `DataSource` for details on the properties to use.

2.2.3. `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`.

2.2.4. `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.RemoteCommitProviderClass`

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

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

Resource adaptor config-property: `RemoteCommitProviderClass`

Default: none

Description: The name of the class to use for communicating commit information among JVMs. Must implement `com.solarmetric.kodo.runtime.event.RemoteCommitProvider`.

2.2.6. `com.solarmetric.kodo.RemoteCommitProviderProperties`

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

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

Resource adaptor config-property: `RemoteCommitProviderProperties`

Default: none

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

2.2.7. `com.solarmetric.kodo.DefaultDataCacheTimeout`

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

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

Resource adaptor config-property: `DefaultDataCacheTimeout`

Default: 0.0

Description: The number of seconds that data in the data cache is valid for. A value of 0 or less means that by default, cached data does not time out.

2.2.8. 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.

2.2.9. 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.

2.2.10. 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.

2.2.11. com.solarmetric.kodo.FetchGroups

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

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

Resource adaptor config-property: `FetchGroups`

Default: `none`

Description: A comma-separated list of fetch group names that are to be loaded when loading objects from a data store.

2.2.12. com.solarmetric.kodo.FlushBeforeQueries

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

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

Resource adaptor config-property: `FlushBeforeQueries`

Default: `with-connection`

Possible values:

- `true`
- `false`
- `with-connection`

Possible values:

Description: Controls whether or not Kodo should automatically flush modifications to the data store before executing queries. Kodo will only flush modifications if `javax.jdo.option.IgnoreCache` is set to `false` (not the default) and this property is set to `true` or `with-connection` and Kodo detects that the changes made in the current transaction may be involved in the query to be executed.

If `FlushBeforeQueries` is set to `with-connection`, then Kodo will only automatically flush if the current `PersistenceManager` already has a reference to a connection. This means that Kodo will sacrifice query performance in order to not obtain dedicated resources. This option is useful if you use long-lived optimistic transactions and want to ensure that these long-lived transactions will not consume database resources for the life cycle of the transaction. The behavior of Kodo when `with-connection` is used is dependent upon the current transactional mode and the setting of `ConnectionRetainMode`.

If `IgnoreCache` is `false`, Kodo detects that there may be modifications, and this property is set to `false`, Kodo will perform queries in-memory.

Below is a table describing the behavior of automatic flushing in various different situations. In all these situations, flushing will only occur if Kodo detects that you have made modifications in the current transaction to instances of types that are in the current query's access path.

Table 2.1. Kodo Automatic Flush Behavior

	FlushBeforeQueries = false	FlushBeforeQueries = true	FlushBeforeQueries = with-connection; ConnectionRetainM ode = on-demand	FlushBeforeQueries = with-connection; ConnectionRetainM ode = transaction or persistence-manager
IgnoreCache = true	no flush	no flush	no flush	no flush
IgnoreCache = false; no tx active	no flush	no flush	no flush	no flush
IgnoreCache = false; datastore tx active	no flush	flush	flush	flush
IgnoreCache = false; optimistic tx active	no flush	flush	no flush unless flush() has already been invoked	flush

2.2.13. 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

2.2.14. `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`.

2.2.15. `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.

2.2.16. `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`.

2.2.17. `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.

2.2.18. `com.solarmetric.kodo.QueryCacheClass`

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

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

Resource adaptor config-property: `QueryCacheClass`

Default: none

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

2.2.19. `com.solarmetric.kodo.QueryCacheProperties`

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

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

Resource adaptor config-property: `QueryCacheProperties`

Default: none

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

2.2.20. `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.

2.2.21. `com.solarmetric.kodo.ResultListClass`

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

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

Resource adaptor config-property: `ResultListClass`

Default: none

Description: The name of the class to use as the List implementation that holds Query results. By default, this value will be dependant on whether the JDBC driver supports scrollable cursors, but can be overridden by explicitly setting this property. The class must implement the `com.solarmetric.kodo.runtime.objectprovider.CustomResultList` interface. The following built-in implementations are available:

- **`com.solarmetric.kodo.runtime.objectprovider.EagerResultList`:** This implementation will instantiate all the results immediately, and free the underlying datastore resource. This is the default that will be used when the underlying JDBC driver does not support scrolling ResultSets.

- **com.solarmetric.kodo.runtime.objectprovider.OnDemandForwardResultList:** This implementation is never used by default. It can be used when scrolling ResultSets incur a large performance penalty, but support for large ResultSets is still desired. It instantiates requested elements on demand in a forward-only fashion. If the UseWindow configuration property is set to true in the com.solarmetric.kodo.ResultListProperties property, then the list will only hold onto a window-full of references to objects, and will drop references to any lower indices in the list. The window size is defined by the fetch batch size setting of the PersistenceManager when the list is created.

Warning

Since this ResultList implementation is instantiated in a forward-only fashion, the size of the Collection will be reported incorrectly until the all of the results have been completely instantiated. This can lead to problems with code that relies on the correct size of a Collection, including many of the contracts defined by the java.util.Collection interface. Until the results are completely instantiated, the result of OnDemandForwardResultList.size() will be java.lang.Integer.MAX_VALUE.

the fetch batch size setting of the PersistenceManager when the list is created.

- **com.solarmetric.kodo.impl.jdbc.runtime.LazyResultList:** This implementation will keep open the datastore resource in order to provide random access to potentially large result lists. It is used by default when the JDBC driver supports scrolling ResultSets.

2.2.22. com.solarmetric.kodo.ResultListProperties

Property name: com.solarmetric.kodo.ResultListProperties

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

Resource adaptor config-property: ResultListProperties

Default: none

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

2.2.23. com.solarmetric.kodo.TransactionCacheClass

Property name: com.solarmetric.kodo.TransactionCacheClass

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

Resource adaptor config-property: TransactionCacheClass

Default: com.solarmetric.kodo.runtime.FifoStateManagerSet

Description: The name of the class to use to store persistence-capable objects involved in a transaction. Must implement com.solarmetric.kodo.runtime.StateManagerSet. See the StateManagerSet documentation for possible values.

2.2.24. com.solarmetric.kodo.TransactionCacheProperties

Property name: com.solarmetric.kodo.TransactionCacheProperties

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

Resource adaptor config-property: TransactionCacheProperties

Default: none

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

2.2.25. `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.

2.2.26. `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.

2.2.27. `com.solarmetric.kodo.TransactionMode`

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

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

Resource adaptor config-property: `TransactionMode`

Default: `direct`

Description: When set to "xa", the `PersistenceManager` will not pass `commit()` requests directly to the data store, but will allow the current XA-compliant global transaction to handle the commit. This should only be used in managed environments where `XADataSources` are being used.

2.2.28. `com.solarmetric.kodo.ConnectionFactory2Properties`

Property name: `com.solarmetric.kodo.ConnectionFactory2Properties`

Configuration API: `com.solarmetric.kodo.conf.Configuration.getConnectionFactory2Properties`

Resource adaptor config-property: ConnectionFactory2Properties

Default: none

Description: Enables the configuration of a ConnectionFactory2 without binding it into JNDI. A space-separated list of bean setters for the com.solarmetric.kodo.impl.jdbc.ConnectionFactoryConfiguration instance to configure the ConnectionFactory2, which is used for obtaining datastore identities.

Example 2.3. Specifying ConnectionFactory2 Properties

```
com.solarmetric.kodo.ConnectionFactory2Properties: \  
  ConnectionURL=jdbc:oracle:thin:@DBHOST:1521:DBNAME \  
  ConnectionDriverName=oracle.jdbc.driver.OracleDriver \  
  ConnectionUserName=USER \  
  ConnectionPassword=PASS
```

ConnectionFactory2, which is used for obtaining datastore identities.

2.2.29. com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode

Property name: com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode

Configuration API: com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getConnectionRetainMode

Resource adaptor config-property: ConnectionRetainMode

Default: on-demand

Description: Controls the association between a PersistenceManager and a JDBC Connection. Possible values:

- **on-demand:** obtain connections as necessary, releasing them as soon as they are no longer needed. When using pessimistic transactions, connections are held for the duration of transactions, so that locks can be obtained correctly -- the behavior is the same as if transaction had been used. If a request for a connection is made while a connection is in use, the same connection will be returned, and an internal reference counter will be incremented. This means that a PM will never use more than one connection (ignoring any connections used by the SequenceFactory). This behavior differs from pre-2.5.3 versions of Kodo. If the old-style on-demand connection retain mode is needed, consider using the legacy-on-demand mode (see below).
- **transaction:** obtain connections on a per-transaction basis. That is, a connection is obtained and used for the duration of each transaction. Nontransactional database accesses will obtain connections on-demand.
- **persistence-manager:** obtain connections on a per-PersistenceManager basis, releasing them when the PersistenceManager is closed. All database accesses will use the same connection.
- **legacy-on-demand:** just like on-demand, except that if multiple nested requests for a connection occur, multiple connections will be fetched from the pool. This is inefficient, and can possibly result in race conditions if the pool size is sufficiently small. This setting should therefore be avoided, and it is likely that this setting will be removed in a future release of Kodo.

Description: Controls the association between a PersistenceManager and a JDBC Connection. Possible values:

2.2.30. com.solarmetric.kodo.impl.jdbc.ConnectionTestTimeout

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

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

2.2.31. **com.solarmetric.kodo.impl.jdbc.DefaultClassMappingClass**

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

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

2.2.32. **com.solarmetric.kodo.impl.jdbc.DefaultSubclassProviderClass**

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

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

2.2.33. **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.

2.2.34. **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`

abstract away the idiosyncrasies of an individual database vendor. Each supported database has its own dictionary: 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.

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

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

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBConfiguration.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 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`.

The `MySQLDictionary` understands the following properties:

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

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

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

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

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

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.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 when the schematool is run.

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. Possible values:

- `com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory`. A `SequenceFactory` capable of generating globally unique IDs.
- `com.solarmetric.kodo.impl.jdbc.schema.ClassDBSequenceFactory`. A class-sensitive `SequenceFactory`. This takes the same options as `DBSequenceFactory`.
- `com.solarmetric.kodo.impl.jdbc.schema.TrueSequenceFactory`. A `SequenceFactory` that uses database sequences to obtain globally unique values. This will only work for databases that support direct access and manipulation to database sequences.
- `com.solarmetric.kodo.impl.jdbc.schema.ClassSequenceFactory`. A class-sensitive `SequenceFactory` that uses database sequences to obtain values. This will only work for databases that support direct access and manipulation to database sequences.
- `com.solarmetric.kodo.impl.jdbc.schema.AutoIncrementSequenceFactory`. A class-sensitive `SequenceFactory` that uses auto-incrementing primary key columns to obtain values. This will only work with database dictionaries that support auto-incrementing identity columns.

included in the source directory of your Kodo Installation. Possible values:

2.2.38. `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.

2.2.39. `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`.

2.2.40. `com.solarmetric.kodo.impl.jdbc.SQLExecutionListenerClass`

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

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

Resource adaptor config-property: `SQLExecutionListenerClass`

Default: `none`

Description: A single implementation that will be installed as a custom `SQLExecutionListener` to listen to all JDBC activity. `com.solarmetric.kodo.impl.jdbc.SQLExecutionListener`.

2.2.41. `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: `70`

Description: The size of the `PreparedStatement` cache that is maintained in the `DataSource` implementation. This value is global to the data source, rather than per-connection.

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

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

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

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

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

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

classes are referenced, allowing the developer to bypass the `schematool` step.

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

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

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

are:

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

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

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

2.2.46. `com.solarmetric.kodo.impl.jdbc.UseSQL92Joins`

Property name: `com.solarmetric.kodo.impl.jdbc.UseSQL92Joins`

Configuration API: `com.solarmetric.kodo.impl.jdbc.JDBCConfiguration.getUseSQL92Joins`

Resource adaptor config-property: `UseSQL92Joins`

Default: `database`

Description: Set this property to `true` to make queries use SQL 92-style joins, including left outer joins where appropriate. Note that some databases do not support the SQL 92 standard, but do have a native outer join syntax. For these databases, set the value of this property to `database`. For databases that only support traditional joins, set this property to `false`. This property can also be set on a query-by-query basis; see the `com.solarmetric.kodo.impl.jdbc.query.JDBCQuery` Javadoc for details.

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

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

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

2.2.48. `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.

2.2.49. `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.

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

2.3. Logging Framework

Logging is very important for debugging and identifying performance hot spots in an application, as well as getting a sense of how Kodo operates. Using logging is essential for developing any persistent classes with custom object-relational mapping extensions, since observing the SQL that is generated will assist in quickly identify any misconfigurations in the JDO metadata file. Kodo provides a very flexible logging system that integrates with many existing runtime systems, such as application servers and servlet runners.

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.

Warning

Logging can have a very serious performance impact on Kodo. Disable verbose logging (such as logging of SQL statements) before running any performance tests. It is advisable to limit or disable logging completely for a production system.

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.Performance`: Information about possible performance optimizations will be sent to the trace level of 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 `trace` channel. It is useful to enable this channel if you are curious about the exact SQL that Kodo issues to the data store.

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.

useful to enable this channel if you are curious about the exact SQL that Kodo issues to the data store.

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

2.3.1. Disabling Logging

Disabling logging can be useful for performance analysis without any I/O overhead or to reduce verbosity at the console. To do this, set the `org.apache.commons.logging.Log` to `org.apache.commons.logging.impl.NoOpLog`. To do this via command line: However, disabling logging

`java -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.NoOpLog mypkg.MyClass`
`org.apache.commons.logging.impl.NoOpLog`. To do this via command line: However, disabling logging permanently will cause all error messages to be consumed. So, we recommend using one of the more sophisticated mechanisms described below.

Note

Versions of the Apache Commons Logging prior to 1.0.3 ignore the `org.apache.commons.logging.Log` system property. To resolve this, upgrade to a more recent version of the logging APIs.

2.3.2. Logging using Apache Log4J

When Apache Log4J jars are present, the Commons Logging package will use it by default. In a standalone application, logging levels are controlled by a resource named `log4j.properties`, which should be available as a top-level resource (either at the top level of a jar file, or in the root of one of the `CLASSPATH` environment variable). When deploying to a web or EJB application server, Log4J configuration is often performed in a `log4j.xml` file instead of in a properties file. For further details on configuring Log4J, please see the Log4J Manual.

Following are example `log4j.properties` files for configuring logging levels for Kodo.

Example 2.4. 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 2.5. 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 2.6. 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
```

2.3.3. Logging using JDK 1.4 java.util.logging

When using JDK 1.4 or higher, the built-in logging package provided by the `java.util.logging` package will be used. For details on configuring the built-in logging system, please see the [Java Logging Overview](#).

By default, JDK 1.4's logging package looks in the `JAVA_HOME/lib/logging.properties` file for logging configuration. This can be overridden with the `java.util.logging.config.file` system property. E.g., to run using a custom logging.properties file, the following command could be issued: **java -Djava.util.logging.config.file=mylogging.properties com.company.MyClass**

Example 2.7. Example logging.properties file

```
# Specify the handlers to create in the root logger
# (all loggers are children of the root logger)
# The following creates two handlers
handlers=java.util.logging.ConsoleHandler, java.util.logging.FileHandler

# Set the default logging level for the root logger
.level=ALL

# Set the default logging level for new ConsoleHandler instances
```

```
java.util.logging.ConsoleHandler.level=INFO

# Set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level=ALL

# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# Set the default logging level for all Kodo logs to INFO
com.solarmetric.kodo.impl.jdbc.SQL.level=INFO
com.solarmetric.kodo.impl.jdbc.JDBC.level=INFO
com.solarmetric.kodo.impl.jdbc.Schema.level=INFO
com.solarmetric.kodo.Performance.level=INFO
com.solarmetric.kodo.Metadata.level=INFO
com.solarmetric.kodo.Enhance.level=INFO
com.solarmetric.kodo.Query.level=INFO
com.solarmetric.kodo.Runtime.level=INFO
```

2.3.4. Logging using Simple Log

When a version of Java lower than 1.4 is being used, and Log4J libraries are not located in the CLASSPATH, then the commons logging package will fall back to using its built-in simple logging system, using the class

`org.apache.commons.logging.impl.SimpleLog`. This system is controlled by the properties in the `simplelog.properties` resource, or else by various system properties, as specified at the SimpleLog Javadoc.

Example 2.8. Example `simplelog.properties` file

```
# By default, we will log messages at "warn" or higher
org.apache.commons.logging.simplelog.defaultlog=warn

# formatting options
org.apache.commons.logging.simplelog.showShortLogname=true
org.apache.commons.logging.simplelog.showdatetime=true

# Set the default logging level for all Kodo logs to "info"
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.impl.jdbc.SQL=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.impl.jdbc.JDBC=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.impl.jdbc.Schema=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.Performance=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.Metadata=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.Enhance=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.Query=info
org.apache.commons.logging.simplelog.log.com.solarmetric.kodo.Runtime=info
```

2.3.5. Logging using a Custom Log

If it is ever the case where none of the built-in logging packages are suitable for an application, the logging system can be configured to use a custom logging class. This can be used, for example, to integrate with a proprietary logging framework used by some applications servers, or for logging to a graphical component for GUI applications.

Custom logging can be accomplished by writing an implementation of the `org.apache.commons.logging.LogFactory` class, and setting the `org.apache.commons.logging.LogFactory` system property to contain the name of the custom class.

Example 2.9. Example custom logging class

```
public class CustomLoggingExample
    extends org.apache.commons.logging.impl.LogFactoryImpl
{
    public org.apache.commons.logging.Log getInstance (String name)
    {
        // return a simple extension of SimpleLog that will log
        // everything to the System.err stream.
        return new org.apache.commons.logging.impl.SimpleLog (name)
        {
            /**
             * In our example, all log levels are enabled.
             */
            protected boolean isLevelEnabled (int logLevel)
            {
                return true;
            }

            /**
             * Just send everything to System.err
             */
            protected void log (int type, Object message, Throwable t)
            {
                System.err.println ("CUSTOM_LOG: " + type + ": "
                    + message + ": " + t);
            }
        };
    }
}
```

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.

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

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.ApplicationIdTool com.solarmetric.examples.Person
java com.solarmetric.kodo.enhance.ApplicationIdTool -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 `jdock` script or via its Java class, `com.solarmetric.kodo.enhance.JDOEnhancer`. The preferred method of enhancement is the `jdock` 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
jdock -licenseKey xxx-yyy-zzz com/solarmetric/examples/*.jdo
jdock com.solarmetric.examples.Person Employee.class com/solarmetric/examples/Project.jdo
```

3.3. Auto-Generating Classes from a Schema

Kodo JDO's reverse mapping tool generates persistent class definitions, complete with JDO metadata and Kodo mapping extensions, from an existing database schema. 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)

databases:

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

1. 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`. Specify the file to write to with the `-file` or `-f` command-line flag. For example:

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

You can limit the schemas and/or tables the schema generator looks at with the `-schemas` or `-s` command-line argument. This argument expects a comma-separated list of schema and table names. Specify table names 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 -file schema.xml -schemas BUSOBSJS,GENERAL.ADDRESS,.SYSTEM_INFO
```

is in (or it may not have a schema).

2. 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 XML format of 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.

3. 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 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:

- `-schemas/-s <schema and table names>` : A comma-separated list of schema and table names to reverse map, if no XML schema file is supplied. The format of the list is the same as that described for the R&D schema generator above.
- `-package/-pkg <package name>`: The package name of the generated classes. If no package name is given, the generated code will not contain package declarations.
- `-directory/-d <output directory>` : The path to the directory to output all generated code and metadata to. If the directory does not match the package of a class, the package structure will be created beneath this directory. Defaults to the current directory.
- `-useSchemaName/-sn <true/t | false/f>` : 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/-fkn <true/t | false/f>`: 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.
- `-nullableAsObject/-no <true/t | false/f>` : 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/-pkj <true/t | false/f>` : 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.

- `-useDatastoreIdentity/-ds <true/t | false/f>`: Set to `true` to use datastore JDO identity for tables that have single numeric primary key columns. The tool typically uses application identity for all generated classes.
- `-oneToManyRelations/-omr <true/t | false/f>`: Set to `false` to prevent the creation of inverse one-many relations for all one-one relations.
- `-metadata/-md <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.

- `-customizerClass/-cc <class name>` : The full class name of a `com.solarmetric.rd.kodo.impl.jdbc.meta.ReverseCustomizer` customization plugin. If you do not specify a reverse customizer of your own, the system defaults to a `PropertiesReverseCustomizer`. This customizer allows you to specify simple customization properties in the properties file given with the `-customizerProperties` flag below. We present the available property keys in the below.
- `-customizerProperties/-cp <properties file or resource>`: The path or resource name of a properties file to pass to the reverse customizer on initialization.
- `-customizer./-c.<property name> <property value>`: The given property name will be matched with the corresponding Java bean property in the specified reverse customizer, and set to the given value.
- `-codeFormat./-cf.<property name> <property value>`: Code formatting properties. We enumerate code formatting options below.

Example 3.3. Using the Reverse Mapping Tool

```
rd-reversemappingtool -package com.xyz -directory ~/src 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, and `<package-name>.mapping` file. 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.

4. 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 jdo.mapping
generated classes before the import:
```

You can now delete the mapping file if desired.

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

3.3.1. Customizing Reverse Mapping

The reverse mapping process can be customized with the `com.solarmetric.kodo.impl.jdbc.meta.ReverseCustomizer` plugin interface. See the class Javadoc for details on the hooks this interface provides. Specify the concrete plugin implementation to use with the `-customizerClass/-cc` command-line flag, described in the preceding section.

By default, the reverse mapping tool uses a `com.solarmetric.kodo.impl.jdbc.meta.PropertiesReverseCustomizer`. This customizer allows you to perform relatively simple customizations through the properties file named with the `-customizerProperties` tool flag. The customizer recognizes the following properties:

- `<class name>.rename <new class name>`: Override the given tool-generated class name with a new value. Use full class names, including packages. You are free to rename a class to a new package. Specify a value of `none` to reject the class and leave the corresponding table unmapped.
- `<class name>.identity <datastore | identity class name>`: Set this property to `datastore` to use datastore identity for the named class, rather than the default application identity. Any value other than `datastore` will be considered a new class name for the generated application identity class. Give full class names, including packages. You are free to change the package of the identity class this way. If the persistent class has been renamed, use the new class name for this property key. Remember that datastore identity requires a table with a single numeric primary key column.
- `<class name>.<field name>.rename <new field name>`: Override the tool-generated field name with the given one. Use the field owner's full class name in the property key. If the field owner's class was renamed, use the new class name. The property value should be the new field name, without the preceding class name. Use a value of `none` to reject the generated mapping and remove the field from the class.
- `<class name>.<field name>.type <field type>`: The type to give the named field. Use full class names. If the field or the field's owner class has been renamed, use the new name.
- `<class name>.<field name>.value`: The initial value for the named field. The given string will be placed as-is in the generated Java code, so be sure to add quotes to strings, etc. If the field or the field's owner class has been renamed, use the new name.

tool flag. The customizer recognizes the following properties:

All property keys are optional; if not specified, the customizer keeps the default value generated by the reverse mapping tool.

Example 3.4. Customizing Reverse Mapping with Properties

```
reversemappingtool -pkg com.xyz -cp custom.properties schema.xml
```

Example `custom.properties`:

<code>com.xyz.TblMagazine.rename:</code>	<code>com.xyz.Magazine</code>
<code>com.xyz.TblArticle.rename:</code>	<code>com.xyz.Article</code>
<code>com.xyz.TblPubCompany.rename:</code>	<code>com.xyz.pub.Company</code>
<code>com.xyz.TblSysInfo.rename:</code>	<code>none</code>
<code>com.xyz.Magazine.allArticles.rename:</code>	<code>articles</code>
<code>com.xyz.Magazine.articles.type:</code>	<code>java.util.Collection</code>

```
com.xyz.Magazine.articles.value:      new TreeSet()
com.xyz.Magazine.identity:            datastore

com.xyz.pub.Company.identity:         com.xyz.pub.CompanyId
```

3.3.2. Formatting Reverse Mapping Code

The reverse mapping tool accepts a set of command-line flags for formatting its output to match your coding style. All code formatting flags can begin with either the `codeFormat` or `cf` prefix.

- `-codeFormat./-cf.tabSpaces <spaces>` : The number of spaces that make up a tab, or 0 to use tab characters. Defaults to using tab characters.
- `-codeFormat./-cf.spaceBeforeParen <true/t | false/f>`: Whether or not to place a space before opening parentheses on method calls, if statements, loops, etc. Defaults to `false`.
- `-codeFormat./-cf.spaceInParen <true/t | false/f>`: Whether or not to place a space within parentheses; i.e. `method(arg)`. Defaults to `false`.
- `-codeFormat./-cf.braceOnSameLine <true/t | false/f>`: Whether or not to place opening braces on the same line as the declaration that begins the code block, or on the next line. Defaults to `true`.
- `-codeFormat./-cf.braceAtSameTabLevel <true/t | false/f>`: When the `braceOnSameLine` option is disabled, you can choose whether to place the brace at the same tab level of the contained code.
- `-codeFormat./-cf.scoreBeforeFieldName <true/t | false/f>`: Whether to prefix an underscore to names of private member variables.
- `-codeFormat./-cf.linesBetweenSections <lines>` : The number of lines to skip between sections of code. Defaults to 2.

All code formatting flags can begin with either the `codeFormat` or `cf` prefix.

Example 3.5. Code Formatting with the Reverse Mapping Tool

```
rd-reversemappingtool -package com.xyz -codeFormat.spaceBeforeParen true -schemas BUSOBSJS,OBJS
```

3.3.3. 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 | double | float | integer | java_object |
    longvarbinary | longvarchar | null | numeric | other |
    real | ref | smallint | struct | time | timestamp |
    tinyint | varbinary | varchar) #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.

categories:

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`

4.1.1. 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.

4.1.2. 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.

4.1.3. value-dependent

This extension is equivalent to the `element-dependent` extension above, but is used for map values rather than collection elements.

4.1.4. 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.

4.2.1. 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

4.2.2. 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 and the class being described is a least-derived type, Kodo will add its own primary key column, usually named `JDOIDX`. If the `pk-column` extension is unspecified and the class being described extends from another persistence-capable type, then Kodo will use the primary key column name defined by the superclass.

4.2.3. 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.

4.2.4. 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`. If you specify `none` and try to use Kodo with subclasses, Kodo will throw an exception when it attempts to load data.

4.2.5. 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.

4.2.6. 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.

4.2.7. 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.

4.2.8. 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`.

4.2.9. data-cache-name

The name of the cache that will be used to hold instances of this persistent class. See the documentation on specifying a non-default DataCache.

4.2.10. cache-timeout

This optional extension designates the time in seconds that an instance of this class's data is valid for in the the PersistenceManagerFactory cache. Leaving it unset or setting it to a value less than or equal to 0 will disable timeouts for this class. Setting this to a positive decimal value will cause data of this type loaded into the cache to be deemed invalid after `cache-timeout` seconds have passed. Defaults to 0.

4.2.11. sequence-factory-class

This optional extension configures a custom SequenceFactory for the current persistent type. This is only valid in the least-derived type in an inheritance hierarchy that uses datastore identity. If this extension is unset, the system-wide SequenceFactory specified in the SequenceFactoryClass property will be used.

If the `sequence-factory-properties` is specified, then the value of that extension will be used when configuring the sequence factory.

4.2.12. sequence

The name of the sequence for generating primary keys. Used in conjunction with the `sequence-factory-properties` extension.

4.2.13. sequence-factory-properties

This optional extension defines the configuration properties to use when creating a SequenceFactory for this class. See the `sequence-factory-class` for details.

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.

4.3.1. 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.

4.3.2. 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.

4.3.3. value-dependent

This extension is equivalent to the `dependent` extension above, but applies to first class map values.

4.3.4. 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*).

4.3.5. 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.

4.3.6. 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.

4.3.7. 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.

4.3.8. 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.

4.3.9. 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.

4.3.10. 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.

4.3.11. 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.

4.3.12. key-column

This extension is applicable only to map fields. It specifies the column of the secondary table that holds the key values.

4.3.13. 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.

4.3.14. 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`.

4.3.15. 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.

4.3.16. <tablename>-pk-column

This extension specifies the primary key column to use for the table `tablename`. It is a valid extension to primary-key fields (those fields with the JDO `primary-key` field-level attribute set to `true`).

4.3.17. fetch-group

`fetch-group`: This optional extension stores the name of the custom fetch group in which this field should be included. Zero or one `fetch-group` extensions may be specified. The field-level `default-fetch-group` attribute must be set to `false` if this extension is present. See the fetch groups documentation for more information.

4.3.18. custom-mapping

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

4.3.19. externalizer

The method to invoke in order to externalize the field to a `String`, for storing as a `String`. The method can be a non-static method to invoke on the field value, or a static method that will take the field value as a parameter. See the Storing Second Class Objects via Stringification documentation.

4.3.20. factory

The method to invoke in order to instantiate the field to a `String`, when storage is performed via stringification. The

method should be a static method that takes a single String argument. When unset, the String constructor of the field type will be invoked. See the [Storing Second Class Objects via Stringification](#) documentation.

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.

4.5.1. 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>
    <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>

```

4.5.2. Mapping One to Many relations

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

COMPANIES			
CID	NAME		
1	SolarMetric		
2	Cisco		
3	Sun Microsystems		
...			

EMPS			
EMP_ID	COMPID	FNAME	LNAME
1	1	Abe	White
2	1	Marc	Prud'hommeaux
3	1	Patrick	Linksey
4	3	Bill	Joy
...			

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

```

package com.solarmetric.examples;

...

public class Employee
{
    private String    firstName;
    private String    lastName;
    private Company   company;

    ...
}

public class Company
{
    private String    name;
    private Set       employees;

    ...
}

```

The metadata for this mapping will take advantage of the "inverse" extension, which tells Kodo to skip the use of an intermediate cross-reference table and instead map directly to the traditional one-to-many mechanism commonly used in relational database schema design:

```

<jdo>
    <package name="samples.ormmapping.oneToMany">
        <class name="Company">

```

```

<extension vendor-name="kodo" key="table" value="COMPANIES"/>
<extension vendor-name="kodo" key="pk-column" value="CID"/>
<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="employees">
  <collection element-type="Employee"/>
  <extension vendor-name="kodo" key="inverse" value="company"/>
</field>
</class>

<class name="Employee">
  <extension vendor-name="kodo" key="table" value="EMPS"/>
  <extension vendor-name="kodo" key="pk-column" value="EMP_ID"/>
  <extension vendor-name="kodo" key="lock-column" value="none"/>
  <extension vendor-name="kodo" key="class-column" value="none"/>
  <field name="firstName">
    <extension vendor-name="kodo" key="data-column" value="FNAME"/>
  </field>
  <field name="lastName">
    <extension vendor-name="kodo" key="data-column" value="LNAME"/>
  </field>
  <field name="company">
    <extension vendor-name="kodo" key="data-column" value="COMPID"/>
  </field>
</class>
</package>
</jdo>

```

4.5.3. 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. Additionally, when using datastore identity, the primary key column names can only be changed on a per-class basis -- a subclass can designate a different primary key column than its parent, but if a single subclass maps different fields to multiple tables, all those tables must share the same primary key column name. This restriction does not apply to classes that use application identity.

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: Each additional

- `-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.

by all Kodo JDO tools, the reverse mapping tool recognizes the following command line flags: 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
```

Standard usage example:

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

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

connectivity: 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
```

`javax.jdo.option.ConnectionURL` property will be unused.

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 feature 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 `ResultSets` 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: Any additional arguments to

- *-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!

actions are:

- *-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.

arguments defined by the configuration framework. It also accepts the following flags: 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. Event Notification Framework

6.4.1. Transaction Event Notification

Kodo provides callbacks for a number of `PersistenceManager`-related events. These callbacks provide a mechanism for application code to perform application-specific coding at various key transaction lifecycle points: transaction begin, transaction rollback, and transaction commit.

To receive callbacks, implement one or more of the transaction event callback interfaces and register it with the `PersistenceManagerImpl`s that the callback should be associated with:

```
import com.solarmetric.kodo.runtime.*;
import com.solarmetric.kodo.runtime.event.*;
import java.util.*;
```

```
TransactionCommitListener tl = new TransactionCommitListener ()
{
    public void transactionCommitted (PersistenceManagerImpl pm, Set added,
        Set updated, Set deleted)
    {
        // do something with the lists of added, updated, and
        // deleted StateManagerImpl objects
    }
};

((PersistenceManagerImpl) pm).registerTransactionListener (tl);
```

There are three listener interfaces that can be registered with a `PersistenceManagerImpl`:

- **TransactionCommitListener**

The `TransactionCommitListener` interface provides a mechanism for receiving notification before a commit begins and upon successful completion of a transaction. This notification includes collections of `StateManagerImpl` objects involved in the transaction.

- **TransactionStartListener**

The `TransactionStartListener` interface provides a method for receiving notification of a `Transaction.begin()` invocation.

- **TransactionRollbackListener**

The `TransactionRollbackListener` interface provides a method for receiving notification of a `Transaction.rollback()` invocation. This is invoked both when `rollback()` is directly invoked and when a transaction is implicitly rolled back due to a commit-time failure.

There are three listener interfaces that can be registered with a `PersistenceManagerImpl`:

6.4.2. Remote Commit Notification

In addition to receiving callbacks when a particular `PersistenceManager` goes through certain life cycle events, it is possible to receive notification when remote transactions commit with Kodo JDO Enterprise Edition. See the remote commit notification documentation for details on how this works.

6.5. 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.kodo.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.

6.6. 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.

The setting of the `com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode` property has an impact on the semantics of this method. If the property is set to `on-demand` (the default) and an optimistic transaction is in progress, then the connections returned by this method will be freshly retrieved from the pool, and may not be the same as the one upon which the transaction will eventually commit. If it's set to `transaction` or `persistence-manager`, then it will be used for the duration of the transaction. The connection returned can safely be released regardless of the

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);
```

will be used for the duration of the transaction. 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: Note that for this example, it is up to you to do all cleaning up. Additionally, you may need to enter a

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

6.7. `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.

6.8. 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.IntegerSubclassProvider"/>

      <!-- 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>
```

`com.acme.object.Bar` have a value of 2:

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.

6.9. Storing Second Class Objects via Stringification

Object fields that are neither instances of `PersistenceCapable` nor one of the default persistent types mandated by the JDO specification (primitives and wrappers, `String`, `Locale`, etc.) are, by default, persisted to blob fields by serializing the object. This has a number of drawbacks:

- serialization can be slow in some cases
- serialized fields can be larger than are necessary in some cases
- serialized fields cannot be queried
- since the database will store the java serialized bytes, other non-java applications are not able to share the data with the JDO application in a meaningful way

Kodo offers the ability to write custom field mappings in order to have complete control over the mechanism with

which fields are stored, queried, and loaded from the datastore. Furthermore, since simple types can often be represented as a string (such as `java.net.URL`), Kodo provides metadata extensions to allow the simple storage of classes via stringification/destringification without having to write any custom code.

The field-level externalizer metadata extension will contain the name of a method that will be invoked to convert the field into a `String` for storage in the database. This extension can take either the name of a non-static method, which will be invoked on the field value, or a static method, which will be invoked with the field value as a parameter. Static methods can be specified using the format `"ClassName.method"`.

The field-level factory metadata extension will contain the name of a method that will be invoked to instantiate the field from the `String` stored in the database. This extension will take a static method name, which will be invoked with the `String` value of the datastore value, and must return an instance of the field type. If this extension is not specified, Kodo will use the constructor of the field type that takes a single `String` argument, or will throw a `JDOFatalException` if no constructor with that signature exists.

Example 6.3. Metadata for persisting field of type `java.lang.Class`

```
<!-- A field of type Class, stored by the class name. It
      will be saved with the getName method, and
      re-instantiated with the static forName factory method. -->
<field name="clas" default-fetch-group="true">
  <extension vendor-name="kodo" key="externalizer" value="getName" />
  <extension vendor-name="kodo" key="factory" value="forName" />
</field>
```

Example 6.4. Metadata for persisting field of type `java.net.URL`

```
<!-- The field of type java.net.URL can be externalized to
      a String with the URL.toExternalForm method. Since we
      do not specify the "factory" extension, the String
      constructor for URL will be used to recreate it.
      Since the constructor does not deal with nulls, we
      specify that the JDO layer should throw an exception
      when we attempt to store a null value for the field. -->
<field name="url" null-value="exception" default-fetch-group="true">
  <extension vendor-name="kodo" key="externalizer" value="toExternalForm" />
</field>
```

Example 6.5. Metadata for persisting field of type `java.net.InetAddress`

```
<!-- The field of type java.net.InetAddress can
      be externalized with the InetAddress.getHostAddress()
      method, and restored with InetAddress.getByName(String).
      Note that we could also store the host name by having
      the externalizer instead by InetAddress.getHostName(), but
      that would result in very slow DNS queries every time the
      object is resored. Nulls are not handled by the
      factory, so we set them to throw an exception at
      the JDO level. Since the factory method might be slow,
      we set the default-fetch-group to be false, so the
      address will not be instantiated until it is needed. -->
<field name="address" null-value="exception"
      default-fetch-group="false">
  <extension vendor-name="kodo" key="externalizer" value="getHostAddress" />
```

```
<extension vendor-name="kodo" key="factory" value="InetAddress.getByName" />
</field>
```

Example 6.6. Metadata for persisting field of type `java.util.TimeZone`

```
<!-- The field of type java.util.TimeZone can be
constructed with the static TimeZone.getTimeZone
factory, and externalized via TimeZone.getID. Note that
that TimeZone is an abstract class; since we have
a static factory method, this isn't a problem. -->
<field name="timeZone" null-value="exception"
default-fetch-group="true">
  <extension vendor-name="kodo" key="externalizer" value="getID" />
  <extension vendor-name="kodo" key="factory" value="TimeZone.getTimeZone" />
</field>
```

Example 6.7. Metadata for persisting field of type `java.io.File`

```
<!-- java.io.File objects are represented by the
externalizer File.getAbsolutePath. File's String
constructor will be used to restore the File from
the String. -->
<field name="file" null-value="exception"
default-fetch-group="true">
  <extension vendor-name="kodo" key="externalizer" value="getAbsolutePath" />
</field>
```

Note that by default, stringification fields are not in the default fetch group, so they will be loaded in a secondary statement when first accessed. This can be overridden by setting the `default-fetch-group` attribute of the field to be `true`.

Note

As with fields that are stored through serialization, it is not possible to perform queries against fields that are stored via stringification. Furthermore, since access to the field is not mediated, changes to the internal state of the field will not be detected by the JDO's `StateManager`, and thus the field will only be marked dirty if the field is explicitly dirtied or reset.

6.10. Primary Key Generation

Kodo provides various mechanisms to control how primary keys are generated when using datastore identity. The control of how keys are obtained is done via the `com.solarmetric.kodo.impl.jdbc.SequenceFactory` implementation specified by the `com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass` property, and parameters of the `SequenceFactory` are controlled via the `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties` property.

Using one of the various `SequenceFactory` implementations allows coarse-grained control over identity assignment while maintaining the simplicity of using datastore identity. Furthermore, it is possible to use the `SequenceFactory` interface to make default assignments to the fields of application identity instances. This enables the utilization of the fine-grained controls over identity that is only possible when using application identity, and still retaining the ability to interact with the database for reliable unique sequence generation. Sequence numbers for application identity classes can be obtained by invoking `JDBCConfiguration.html.getSequenceFactory`.

6.10.1. Using table-based key generation

By default, Kodo uses a single table to hold the current numeric sequence value for assigning primary keys using the `com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory` `SequenceFactory` implementation. The name of the table is, by default, `JDO_SEQUENCE`; this can be changed using the `TableName` parameter of the `DBSequenceFactory`. When a sequence number is first needed for assignment to an object identity instance the a newly persistent object, the `DBSequenceFactory` will obtain the current value of the sequence table and increment it by the value specified by the `Increment` parameter of the the `DBSequenceFactory`. Kodo will then use this pool of sequence numbers to assign values to new persistent instances.

Example 6.8. Properties for configuring `DBSequenceFactory`

```
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass=\
  com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
# The table "MY_SEQUENCE_TABLE" will be used for holding the
# sequence values. A pool of 1,000 sequences will be "checked out" at a
# time, which is suitable for large bulk inserts.
com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties=\
  TableName=MY_SEQUENCE_TABLE \
  Increment=1000
```

pool of sequence numbers to assign values to new persistent instances.

6.10.2. Using Database Sequences for key generation

Most databases support a "native sequence" function, which is a built-in mechanism for obtaining incrementing numbers from a "sequence". For example, in Oracle, a database sequence can be created with a statement like "create sequence mysequence". Sequence values can then be simultaneously obtained and incremented with the statement "select mysequence.nextval from dual". Kodo provides support for this common mechanism of sequence assignment with the `com.solarmetric.kodo.impl.jdbc.schema.ClassSequenceFactory` `SequenceFactory` implementation. Sequence names are defined on a per-class basis in the metadata for the persistent class using the sequence class-level metadata extension.

Example 6.9. Properties for configuring `ClassSequenceFactory`

```
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass=\
  com.solarmetric.kodo.impl.jdbc.schema.ClassSequenceFactory
# Set the name of the table from which we will be obtaining sequences
com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties=TableName=DUAL
```

persistent class using the sequence class-level metadata extension.

Example 6.10. Metadata for configuring `DBSequenceFactory`

```
<?xml version="1.0"?>
<jdo>
  <package name="com.mycompany">
    <class name="MyPersistenceClass">
      <extension vendor-name="kodo" key="sequence" value="my_sequence_name"/>
    </class>
  </package>
</jdo>
```

persistent class using the sequence class-level metadata extension.

6.11. `StateManagerSet` configuration

The default `StateManagerSet` (`com.solarmetric.kodo.runtime.FifoStateManagerSet`) maintains the list of transactional persistence-capable objects in the order that they were dirtied in a transaction. In general, this is a good approach since it allows non-deferred referential integrity constraints to be met. If constraint checking is deferred or is disabled, then it is possible to increase performance by using the `com.solarmetric.kodo.runtime.ClassGroupStateManagerSet`. This class groups the persistence-capable objects that are involved in a transaction by class, so that when they are flushed to the database, Kodo is capable of increasing the amount of batching that can be performed. To use the `ClassGroupStateManagerSet`, set the `com.solarmetric.kodo.TransactionCacheClass` property to `com.solarmetric.kodo.runtime.ClassGroupStateManagerSet`.

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. Datastore Cache

7.1.1. 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.

There are four ways to access data via the JDO APIs: relation traversal, JDOQL queries, direct invocation of `PersistenceManager.getObjectById()`, and iteration over an Extent's iterator. Kodo JDO's cache plugin accelerates three of these mechanisms. It does not provide any caching of Extent iterators. If you find yourself in need of higher-performance extent iteration, consider this workaround.

Table 7.1. Data access methods

Access method	Uses cache
Relation traversal	Yes
JDOQL query	Yes
<code>PersistenceManager.getObjectById()</code>	Yes
Iteration over an Extent	No

need of higher-performance extent iteration, consider this workaround.

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.

Kodo's datastore cache can operate both in a single-JVM environment and in a multi-JVM environment. Multi-JVM caching is achieved through use of the event notification framework. The datastore caching and the distributed event notification frameworks are both bundled with Kodo JDO Enterprise Edition, and are available as optional plug-ins for Kodo JDO Standard Edition.

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

When used in conjunction with a `com.solarmetric.kodo.runtime.event.RemoteCommitProvider`, commit information is communicated to other JVMs via JMS or TCP, and remote caches are invalidated based on this information.

When using a Tangosol Coherence cache plug-in, all remote updating of cache information is delegated to the Coherence cache. See the descriptions of the different distributed caches below for more information.

7.1.2. 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.CacheImpl`, and set the `com.solarmetric.kodo.RemoteCommitProviderClass` to `com.solarmetric.kodo.runtime.event.impl.SingleJVMRemoteCommitProvider`.

To configure the `PersistenceManagerFactory` cache to remain up-to-date in a distributed environment, set the `com.solarmetric.kodo.DataCacheClass` property to `com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl`, and set the `com.solarmetric.kodo.RemoteCommitProviderClass` and `com.solarmetric.kodo.RemoteCommitProviderProperties` appropriately. This process is described in greater depth in the remote event notification documentation.

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`.

A cache timeout value can be specified for a class by setting the `timeout` metadata extension to a positive decimal representing the amount of time in seconds for which a class's data is valid.

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: A

Example 7.1. Pinning an object into the `DataCache`

```
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;
import com.solarmetric.kodo.conf.Configuration;

Configuration conf = ((PersistenceManagerImpl) pm).getConfiguration ();
conf.getDataCache ().pin (JDOHelper.getObjectId (o));
```

`remove()` call is made. To pin an object into memory, obtain a reference to the cache and invoke `pin()` on it: A previously pinned object can later be unpinned by invoking `DataCache.unpin()`:

Example 7.2. Unpinning an object from the `DataCache`

```
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;
import com.solarmetric.kodo.conf.Configuration;

Configuration conf = ((PersistenceManagerImpl) pm).getConfiguration ();
conf.getDataCache ().unpin (JDOHelper.getObjectId (o));
```

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

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

```
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;
import com.solarmetric.kodo.conf.Configuration;

Configuration conf = ((PersistenceManagerImpl) pm).getConfiguration ();
conf.getDataCache ().remove (JDOHelper.getObjectId (o));
```

It is possible for different persistence-capable classes to use different caches. This is achieved by specifying a cache name in a JDO metadata class-level extension: This will cause instances of the `NonDefaultCacheClass` class to

Example 7.4. Specifying a non-default DataCache

```
<jdo>
  <package name="">
    <class name="NonDefaultCacheClass">
      <extension vendor-name="kodo" key="data-cache-name" value="small-cache"/>
    </class>
  </package>
</jdo>
```

name in a JDO metadata class-level extension: This will cause instances of the `NonDefaultCacheClass` class to be stored in a cache named `small-cache`. Currently, Kodo does not provide any direct semantics for simple configuration of named caches. Instead, Kodo creates all caches in the same manner, by using the `DataCacheClass` and `DataCacheProperties` configuration properties. Any additional configuration must take place after initialization by using the `getDataCache(String)` method in `com.solarmetric.kodo.runtime.datacache.DataCacheStoreManager` or by extending the `newDataCache(String)` method in `DataCacheStoreManager`.

7.1.3. Kodo JDO Query Caching

Query caching is enabled by default when datastore caching is enabled. The cache stores the object IDs returned by invocations of the `Query.execute()` methods. When a query is executed, Kodo assembles a key based on the query properties and the parameters used at execution time, and checks for a cached query result. If one is found, the object IDs in the cached result are looked up, and the resultant persistence-capable objects are returned. Otherwise, the query is executed against the database, and the object IDs loaded by the query are put into the cache. The object ID list is not cached until the list returned at query execution time is fully traversed.

The default query cache implementation caches 100 query executions in a least-recently-used cache. This can be changed by setting the cache size in the `QueryCacheProperties` configuration property:

`com.solarmetric.kodo.QueryCacheProperties: CacheSize=1000`. Setting this to 0 will disable query caching. Setting it to -1 will disable query cache flushing, causing the cache to retain all results, only removing query data when the queries are invalidated.

There are certain situations in which the query cache is bypassed:

- Caching is not used for in-memory queries (queries in which the candidates are a collection instead of a class or extent).
- Caching is not used in transactions that have `IgnoreCache` set to `false` and in which modifications to classes in the query's access path have occurred. If none of the classes in the access path have been touched, then cached results are still valid and are used.

- Caching is not used in pessimistic transactions, since Kodo must go to the database to lock the appropriate rows.

Cache results are removed from the cache when instances of classes in a cached query's access path are touched. That is, if a query accesses data in class A, and instances of class A are modified, deleted, or inserted, then the cached query data is dropped from the cache.

Additionally, cache results are removed when an object in the query's result list is evicted from cache due to a timeout or space requirements. This is done because if a significant number of objects in a cached query are not in the data cache, then it is more efficient to go to the database than to pull the objects in from the database one-at-a-time.

It is possible to tell the query cache that a class has been altered. This is only necessary when the changes occur via direct modification of the database outside of Kodo's control. When using one of Kodo's distributed cache

Example 7.5. Notifying the query cache of altered classes

```
import com.solarmetric.kodo.conf.Configuration;
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;
import com.solarmetric.kodo.runtime.datacache.query.QueryCache;

Configuration conf = ((PersistenceManagerImpl) pm).getConfiguration ();
QueryCache cache = conf.getQueryCache ();
Set changed = new HashSet ();
changed.add (A.class);
changed.add (B.class);
cache.notifyChangedClasses (changed);
```

direct modification of the database outside of Kodo's control. When using one of Kodo's distributed cache implementations, it is necessary to perform this in every JVM -- the change notification is not propagated automatically. When using a coherent cache implementation such as Kodo's Tangosol cache implementation, it is not necessary to do this in every JVM (although it won't hurt to do so), as the cache results are stored directly in the coherent cache.

Data can manually be dropped from the cache or pinned into the cache, as well. To do so, you must first create a `QueryKey` for the query invocation in question.

Example 7.6. Dropping or pinning query results in the cache

```
import com.solarmetric.kodo.conf.Configuration;
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;
import com.solarmetric.kodo.runtime.datacache.query.QueryCache;
import com.solarmetric.kodo.runtime.datacache.query.QueryKey;

Configuration conf = ((PersistenceManagerImpl) pm).getConfiguration ();
QueryCache cache = conf.getQueryCache ();

QueryKey key1 = new QueryKey (query, params1);
cache.pin (key1);

QueryKey key2 = new QueryKey (query, params2);
cache.remove (key2);
```

`QueryKey` for the query invocation in question.

Pinning data into the cache instructs the cache to not expire the pinned results when cache flushing occurs.

However, pinned results will be removed from the cache if an event occurs that invalidates the results.

Caching can be disabled on a per-PersistenceManager or per-Query basis:

Example 7.7. Temporarily disabling and enabling query caching

```
import com.solarmetric.kodo.runtime.PersistenceManagerImpl;

// temporarily disable query caching for all queries created from pm
PersistenceManagerImpl pmi = (PersistenceManagerImpl) pm;
pmi.setQueryCacheEnabled (false);

// re-enable caching for a particular query
Query q = pm.newQuery (A.class);
q.setQueryCacheEnabled (true);
```

Caching can be disabled on a per-PersistenceManager or per-Query basis:

Example 7.8. Disabling query caching via configuration properties

Alter the PersistenceManagerProperties property to indicate that query caching should be disabled by default:

```
com.solarmetric.kodo.PersistenceManagerProperties: QueryCacheEnabled=false
```

Caching can be disabled on a per-PersistenceManager or per-Query basis:

7.1.4. Kodo JDO Data Cache Configuration

As mentioned above, the datastore cache can work in conjunction with the event notification framework in order to support multi-JVM cache behavior. See the remote commit provider configuration documentation for more information on this process.

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.

Tangosol cache understands the following properties:

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

Example 7.9. 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.

7.1.5. 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.CacheImpl`. If you want to implement a distributed cache that uses an unsupported method for communications, create an implementation of `com.solarmetric.kodo.runtime.event.RemoteCommitProvider`. This process is described in greater detail in the event notification customization documentation.

7.1.6. 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.
- A `com.solarmetric.kodo.runtime.event.RemoteCommitProvider` class must be specified (via the `com.solarmetric.kodo.RemoteCommitProviderClass` property) in order to use the `com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl`, even when using the cache in a single-JVM mode. When using it in a single-JVM context, the property can be set to `com.solarmetric.kodo.runtime.event.impl.SingleJVMRemoteCommitProvider`.

7.1.7. Known issues and limitations

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

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.

- Extents are not cached. So, if you plan on iterating over a list of all the objects in an extent on a regular basis, you will only benefit from caching if you do so with a query instead:

Example 7.10. Use queries instead of extents

```
Extent extent = pm.getExtent (A.class, false);

// This iterator does not benefit from caching...
Iterator uncachedIterator = extent.iterator ();

// ... but this one does.
Query extentQuery = pm.newQuery (extent);
Iterator cachedIterator = ((Collection) extentQuery.execute ().iterator ());
```

you will only benefit from caching if you do so with a query instead:

7.2. Query Extensions

JDOQL is a powerful, easy-to-use query language, but you may occasionally find it limiting in some way. To circumvent the limitations of JDOQL, Kodo JDO Enterprise Edition allows you to extend queries with new operations. Kodo provides some built-in query extensions, and you can develop your own custom extensions as needed.

7.2.1. Using Query Extensions

First, to enable query extensions, you must set the `com.solarmetric.kodo.EnableQueryExtensions` configuration property to `true`.

Next, you must preface all extensions by `ext:` in your JDOQL filter strings. For example, the following example uses Kodo's built-in `stringContains` extension to search for cities with the substring "art" in their name (e.g. Hartford): Note that it is perfectly OK to chain together extensions. For example, let's modify our search above to be

```
Query q = pm.newQuery (City.class);
q.setFilter ("name.ext:stringContains (\"art\")");
Collection c = (Collection) q.execute ();
```

Hartford): Note that it is perfectly OK to chain together extensions. For example, let's modify our search above to be case-insensitive using Kodo's built-in `toLowerCase` extension:

```
Query q = pm.newQuery (City.class);
q.setFilter ("name.ext:toLowerCase ().ext:stringContains (\"art\")");
Collection c = (Collection) q.execute ();
```

case-insensitive using Kodo's built-in `toLowerCase` extension:

Finally, when using query extensions you must be aware that some SQL-specific extensions can only execute against the database, and cannot be used for in-memory queries (recall that JDO executes queries in-memory when

you supply a candidate Collection rather than an Extent/Class, or when you set the `IgnoreCache` and `FlushBeforeQueries` properties to `false` and you execute a query within a transaction in which you've modified some persistent objects). In the list of built-in query extensions below, you can assume each extension can be executed both in-memory and against the database unless its description says otherwise.

7.2.2. Included Query Extensions

Kodo includes several default query extensions to enhance the power of JDOQL:

- *stringContains*: Tests if the target string contains the given argument. The argument must be a string literal or query parameter.

```
query.setFilter ("stringField.ext:stringContains (\"foo\")");
```

query parameter.

- *toLowerCase*: Switches the target string to lower case. Equivalent to Java's `String.toLowerCase ()` method.

```
query.setFilter ("stringField.ext:toLowerCase () == \"foo\")");
```

toLowerCase: Switches the target string to lower case. Equivalent to Java's `String.toLowerCase ()` method.

- *toUpperCase*: Switches the target string to upper case. Equivalent to Java's `String.toUpperCase ()` method.

```
query.setFilter ("stringField.ext:toUpperCase () == \"FOO\")");
```

toUpperCase: Switches the target string to upper case. Equivalent to Java's `String.toUpperCase ()` method.

- *wildcardMatch*: Tests if the target string matches the wildcard pattern given in the argument. Wildcard patterns are strings that use `?` to represent any single character, and `*` to represent any series of 0 or more characters. For example, the wildcard pattern `"*foo?bar"` matches any string that contains the sequence "foo", then a single character, then "bar". The given wildcard argument must be either a string literal or a query parameter.

```
query.setFilter ("stringField.ext:wildcardMatch (\"*foo?bar*\")");
```

character, then "bar". The given wildcard argument must be either a string literal or a query parameter.

- *getColumn*: Places the proper alias for the given column name into the SELECT statement that is issued. This filter cannot be used for in-memory queries. When traversing relations, the column is assumed to be in the primary table of the related type. To get a column of the candidate class, use `this` as the extension target, as shown in the second example below.

```
query.setFilter ("company.address.ext:getColumn (\"JDOIDX\") == 5");
```

```
query.setFilter ("this.ext:getColumn (\"LEGACY_DATA\") == \"foo\")");
```

shown in the second example below.

- *sqlVal*: Embeds the given SQL argument into the SELECT statement that is issued. This filter cannot be used for in-memory queries. Note that the given SQL should evaluate to some value, not a boolean expression. To embed SQL that evaluates to a boolean expression, use the `sqlExp` extension below.

```
query.setFilter ("price < ext:sqlVal (\"(SELECT AVG(PRICE) FROM PRODUCTS)\")");
```

SQL that evaluates to a boolean expression, use the `sqlExp` extension below.

- *sqlExp*: Embeds the given SQL argument into the SELECT statement that is issued. This filter cannot be used for in-memory queries. Note that the given SQL should evaluate to a boolean expression, not some other value. To embed SQL that evaluates to a value, use the `sqlVal` extension above.

```
query.setFilter ("price < 10 || ext:sqlExp (\"(SELECT AVG(PRICE) FROM PRODUCTS)\") > 100");
```

7.2.3. Deprecated Query Extensions

Several of the query extensions defined in earlier versions of Kodo have been deprecated. These include:

- *ext:caseInsStarts*

Requires DB support for UPPER (). Translates to UPPER (column) LIKE UPPER (argument%)

Use `fieldname.toUpperCase ().startsWith ("FOO")` instead.

- *ext:caseInsEnds*

Requires DB support for UPPER (). Translates to UPPER (column) LIKE UPPER (%argument)

Use `fieldname.toUpperCase ().endsWith ("FOO")` instead.

- *ext:caseInsContains*

Requires DB support for UPPER (). Translates to UPPER (column) LIKE UPPER (%argument%)

Use `fieldname.toUpperCase ().ext:stringContains ("FOO")` instead.

- *lit:sqlEmbed*

Embeds argument right into the where clause as-is

Use `ext:sqlExp ("SQL STATEMENT")` instead.

7.2.4. Developing Custom Query Extensions

You can write your own extensions by implementing the `com.solarmetric.kodo.impl.jdbc.query.JDBCFilterListener` interface. View the Javadoc documentation for details. Additionally, the source for all of Kodo's built-in query extensions is included in your Kodo download to get you started. The built-in extensions reside in the `com.solarmetric.kodo.query.listen` and `com.solarmetric.kodo.impl.jdbc.query.listen` packages.

7.2.5. Configuring Query Extensions

There are two ways to register your custom query extensions with Kodo:

- *Registration by properties:* You can register custom query extensions by setting the `com.solarmetric.kodo.QueryFilterListeners` configuration property to a comma-separated list of the full names of your extensions classes. Extensions registered in this fashion must be able to be instantiated via reflection (they must have a public no-args constructor). They must also be thread safe, because they will be shared across all queries.
- *Per-query registration:* You can register query extensions for an individual query through the `QueryImpl.registerListener` method. You might use per-query registration for very specific extensions that do not apply globally.

There are two ways to register your custom query extensions with Kodo:

7.3. Fetch Groups

The JDO specification defines a concept of a default fetch group, but it does not touch upon additional, non-default fetch groups. Kodo JDO extends the JDO specification's fetch group concept to allow multiple fetch groups in a given class. These fetch groups can be used to pool together associated fields in order to provide performance improvements over Kodo JDO's normal fetch group behavior.

7.3.1. 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: In this example, the default fetch group behavior

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

information. Imagine the following class and metadata definitions: In this example, the default fetch group behavior

```
<?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>
```

information. Imagine the following class and metadata definitions: 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.

7.3.2. Kodo JDO Fetch Group Behavior

Kodo JDO improves upon the JDO specification's fetch group configuration options by defining a syntax for declaring extra fetch groups in addition to the default fetch group. A field can be a member of zero or one fetch groups, including the default fetch group. That is, fields in the default fetch group cannot be in an additional fetch group, and a field cannot declare itself a member of more than one fetch group.

When loading an object, fields in these custom fetch groups are not included in the initial select statements, just as if they had been left out of the default fetch group. Upon lazily loading a field, Kodo checks to see if that field declares itself to be a member of a fetch group. If so, Kodo will load all fields in that fetch group.

Additionally, it is possible to configure the `PersistenceManager` to use a particular fetch group or set of fetch groups when loading new objects. When this is done, Kodo loads the default fetch group plus any fields in the set of additional fetch groups specified.

So, a custom fetch group configuration for our `FetchGroupExample` class might look like this: In this example,

```
<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="FetchGroupExample">
      <field name="a" default-fetch-group="true"/>

      <field name="b" default-fetch-group="false">
        <extension vendor-name="kodo" key="fetch-group" value="g1"/>
      </field>

      <field name="c" default-fetch-group="false">
        <extension vendor-name="kodo" key="fetch-group" value="g1"/>
      </field>

      <field name="d" default-fetch-group="false">
        <extension vendor-name="kodo" key="fetch-group" value="g1"/>
      </field>

      <field name="e" default-fetch-group="false">
        <extension vendor-name="kodo" key="fetch-group" value="g2"/>
      </field>

      <field name="f" default-fetch-group="false">
        <extension vendor-name="kodo" key="fetch-group" value="g2"/>
      </field>

      <field name="g"/>
    </class>
  </package>
</jdo>
```

So, a custom fetch group configuration for our `FetchGroupExample` class might look like this: In this example, fields `a` and `g` would be loaded whenever a new object is loaded. (Only the data in column `g` would be loaded -- the object on the other side of this relation would not be loaded since this field is not in the default fetch group.) When lazily loading field `b`, fields `c` and `d` will also be loaded, because they are all in the same fetch group -- `g1`.

If the `PersistenceManager` were configured to load fetch group `g2` when loading new objects, then fields `e` and `f` would be loaded along with the default fetch group members at initial load time. Also, if any relations were traversed, then the fetch groups named would be applied to the loading of the related objects.

7.3.3. Configuring a `PersistenceManager` to load Fetch Groups

As mentioned above, it is possible to configure a `PersistenceManager` to load certain fetch groups when initially loading persistence-capable objects. The simplest way to do this is to use the `com.solarmetric.kodo.FetchGroups` configuration property to specify a comma-separated list of fetch group names to use when loading data. Alternately, the fetch group configuration can be mutated on a per-`PersistenceManager` basis by using the `addFetchGroup(String)`, `addFetchGroups(String[])`, `removeFetchGroup(String)`, `removeFetchGroups(String[])`, `String[] getFetchGroups()`, and `clearFetchGroups()` methods in `com.solarmetric.kodo.runtime.PersistenceManagerImpl`.

7.4. XA Transactions

7.4.1. Overview of XA Distributed Transaction Processing

The X/Open Distributed Transaction Processing (X/Open DTP) model, designed by Open Group (a vendor consortium), defines a standard communication architecture that provides the following:

- Concurrent execution of applications on shared resources
- Coordination of transactions across applications
- Components, interfaces, and protocols that define the architecture and provide portability of applications
- Atomicity of transaction systems
- Single-thread control and sequential function-calling

consortium), defines a standard communication architecture that provides the following:

The X/Open DTP XA standard defines the application programming interfaces that a resource manager uses to communicate with a transaction manager. The XA interfaces enable resource managers to join transactions, to perform two-phase commit, and to recover in-doubt transactions following a failure.

7.4.2. Requirements for using Kodo with XA transactions

Kodo JDO Enterprise Edition support XA-compliant transactions when used in a properly configured managed environment. The following components are required:

- A managed environment that provides an XA compliant `TransactionManager`. Examples of this are application servers such as JBoss and WebLogic.
- Instances of a `javax.sql.XADataSource` for each of the datasources that Kodo will use for communication.

environment. The following components are required:

7.4.3. Configuring Kodo to utilize XA transactions

In order for Kodo to participate in a distributed transaction, the following configuration tasks needs to be accomplished:

- Configure an instance of `javax.sql.XADataSource` to be available from JNDI. The mechanism for configuring data sources is specific to each application server; consult your application server's documentation for details on accomplishing this.

The remainder of this section will assume that you have bound an instance of an Oracle `XADataSource` under

the name `java:/OracleXASource`.

- Configure a separate instance of `javax.sql.DataSource` to be available from JNDI. Kodo needs to have access to a transactional data source that will not be enlisted in an XA transaction for non-transactional access. This is required for things like obtaining database sequence numbers for datastore identity, which should not be part of Kodo's transaction.

The remainder of this section will assume that you have bound an instance of an Oracle `DataSource` under the name `java:/OracleSource`.

- Kodo needs to be aware that it should follow the contracts of a distributed transaction. This is done by specifying the `TransactionMode` attribute to be `"xa"`.
- Kodo must be configured to use the two named data sources. Set the `ConnectionFactoryName` property to be `java:/OracleXASource` and `ConnectionFactory2Name` to `java:/OracleSource`.

For example, if configuring Kodo via a properties file, the following properties would be used:

```
javax.jdo.option.ConnectionFactoryName=java:/OracleXASource
javax.jdo.option.ConnectionFactory2Name=java:/OracleSource
com.solarmetric.kodo.TransactionMode=xa
```

For example, if configuring Kodo via a properties file, the following properties would be used:

7.5. Remote Commit Notification Framework

In addition to receiving callbacks when a particular `PersistenceManager` goes through certain life cycle events, it is possible to receive notification when remote transactions commit. To do this, you must configure the `PersistenceManagerFactory` to use a `RemoteCommitProvider` (see below). Given that a `RemoteCommitProvider` is properly configured, it is possible to register a `RemoteCommitListener` that will be alerted with a list of committed object IDs whenever a transaction on a remote machine successfully commits.

7.5.1. Kodo JDO RemoteCommitProvider Configuration

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

- `Topic`

Default: `topic/KodoCommitProviderTopic`

Description: The topic that the remote commit provider should publish notifications to and subscribe to for notifications sent from other JVMs.

- `TopicConnectionFactory`

Default: `java:/ConnectionFactory`

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

properties. The JMS provider understands the following properties:

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

Example 7.11. Configuring a PersistenceManagerFactory to use a JMS remote commit provider mechanism

```
com.solarmetric.kodo.RemoteCommitProviderClass= \
  com.solarmetric.kodo.runtime.event.impl.JMSRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties= \
  Topic=topic/KodoCommitProviderTopic
```

Note

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.

The TCP `RemoteCommitProvider` has 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 provider can be configured by setting the `com.solarmetric.kodo.RemoteCommitProviderProperties` to contain the appropriate configuration properties. The TCP provider understands the following properties:

- Port

Default: 5636

Description: The TCP port that the provider should listen on for commit notifications.

- Addresses

Default: none

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

properties. The TCP provider understands the following properties:

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

Example 7.12. Configuring a PersistenceManagerFactory to use a TCP remote commit provider mechanism

```
com.solarmetric.kodo.RemoteCommitProviderClass= \
  com.solarmetric.kodo.runtime.event.impl.TCPRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties= \
  Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13
```

the following:

7.5.2. Event Notification Framework Customization

Additional mechanisms for remote event notification can be easily developed by creating an implementation of the `RemoteCommitProvider` interface, possibly by extending the `RemoteCommitProviderImpl` abstract class. For details on particular customization needs, contact SolarMetric at jdosupport@solarmetric.com.

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: The source code for all the ant tasks

- 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.

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

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

It is also possible to specify a `properties` or `propertiesFile` attribute to the `<config>` tag. The `properties` attribute will be used to locate a properties file resource, relative to the current CLASSPATH. The `propertiesFile` can be the relative or absolute path to a properties file.

Example 9.2. Using the properties attribute of the <config> tag

```
<schematool action="refresh" ignoreErrors="true">
  <fileset dir="${basedir}">
    <include name="**/*.jdo" />
  </fileset>
  <config properties="/kodo.properties" />
</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 9.3. 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>
```

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

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 9.4. 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.

The task accepts the following parameters:

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

Example 9.5. 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>
```

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

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 9.6. 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>
```

build.xml as follows:

An example of a commented file is as follows:

Example 9.7. 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>
```

The resulting TestDoclet.jdo metadata file will then look like:

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.

which is used to configure the Kodo installation.

Warning

The Kodo JBuilder OpenTool only works in JBuilder 7 and 8. It will not work in releases of JBuilder prior to version 7.

which is used to configure the Kodo installation.

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: The Pet Shop example allows you to create and delete pets in a

- 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.

directory. To run the sample, do the following: 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-1.0.3.jar`
- `jakarta-commons-lang-1.0.1.jar`
- `xml-apis.jar`
- `jdbc2_0-stdext.jar`
- `jdo1_0.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 seperated 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: The Pet Shop example allows

- 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.

`modules` directory of the IDE home directory. To run the sample, do the following: 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, 2.0, and 2.1 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_1.0.1` 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_1.0.1` directory and place it such that the new directory structure would be: `C:\Eclipse\plugins\com.solarmetric.kodo_1.0.1`.

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.

in mind.

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 ();
```

Applications that utilize Kodo can then obtain a handle to the `PersistenceManagerFactory` as follows:

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.

To deploy to JBoss, you must do the following:

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 `JDOLOCKX`) to determine which column to use for its "lock column". The lock column holds an interger 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, SynchronizeSchema:* Developer options such as logging, 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. *Set IgnoreCache to true, and configure FlushBeforeQueries to flush data automatically before queries :* When the `javax.jdo.option.IgnoreCache` property is set to `false` and `com.solarmetric.kodo.FlushBeforeQueries` is set to `false`, Kodo must evaluate in-memory dirty instances against the datastore values that are returned from a Query. This can sometimes result in Kodo needing to evaluate the entire extent of objects in-memory in order to return the correct query results, which can have drastic performance consequences. If it is appropriate for your application, configuring `FlushBeforeQueries` to automatically flush queries will ensure that this never happens. Setting `IgnoreCache` to `false` will result in a small performance hit even if `FlushBeforeQueries` is `true`, as incremental flushing is not as efficient overall as delaying all flushing to a single operation during commit. This is because incrementally flushing decreases Kodo's ability to maximize statement batching, and increases resource utilization.

Note that the default setting of `FlushBeforeQueries` is `with-connection`, which means that data will be flushed only if a dedicated connection is already in use by the `PersistenceManager`. So, the default value may not be appropriate for you.

7. *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.
8. *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.
9. *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`.
10. *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.

11. *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, optimistic transactions are typically faster than datastore transactions.
12. *Perform nontransactional data reads outside a transaction.*
13. *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 ();
    }
}
```

never necessary to explicitly close these resources, but it is always faster if it is done at the application level.

14. *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. Additionally, as the connection pool size increases, you should also increase the prepared statement pool size, since the prepared statement pool size is global with respect to the connection pool (as opposed to a per-connection size) in Kodo 2.5.
15. *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.
16. *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 only use a given PersistenceManager from a single thread (for example, EJB applications fall into this category), setting this option to false will result in the elimination synchronization overhead, and may result in a modest performance increase.
17. *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`.

18. *Use the `OnDemandForwardResultList`:* By default, Kodo JDO uses a lazy result list implementation. This relies on result sets created with the `ResultSet.TYPE_SCROLL_INSENSITIVE` flag. On some databases (Oracle for example), result sets that support bidirectional scrolling are very memory-intensive, effectively faking bidirectional support in the client tier by loading all data from the beginning to whatever index is requested. This interferes with the behavior of Kodo's `LazyResultList` -- in particular, the method that `LazyResultList` uses to compute the size of the list. So, if you have lots of data and do not want to pay the memory price of loading all the data into memory at one time, and you are using a driver that loads all data when jumping to the end of a result set, then you should use the `OnDemandForwardResultList`. This list is useful in two primary situations -- when it is acceptable to not know the size of the result list as you iterate through it, and when you need to iterate through large amounts of data and are using an inefficient JDBC driver.

If you just want to be able to walk through a result list without incurring memory penalties when computing the size of the underlying result set, set the `DefaultFetchThreshold` property to `-1` to force Kodo to use `ResultSet.TYPE_FORWARD_ONLY` result sets, and set the `ResultListClass` property to `com.solarmetric.kodo.runtime.objectprovider.OnDemandForwardResultList`. The result lists returned from query execution will be lazily loaded and will not know their correct size until the entire list is iterated, and it will be possible to jump to any index in the list.

If you also expect the result list to be quite large and do not want to hold references to all the data at the same time, then you should also set the `UseWindow` configuration property in the `com.solarmetric.kodo.ResultListProperties` property:

```
com.solarmetric.kodo.ResultListClass: com.solarmetric.kodo.runtime.objectprovider.OnDemandForwardResultList
com.solarmetric.kodo.ResultListProperties: UseWindow=true
com.solarmetric.kodo.DefaultFetchThreshold: -1
com.solarmetric.kodo.ResultListProperties property:
```

Note

Bear in mind that for most drivers and situations, the default `LazyResultList` will work fine. It is only in situations where query execution seems to take an inordinately long time or memory usage becomes an issue that the `OnDemandForwardResultList` becomes useful.

`com.solarmetric.kodo.ResultListProperties` property:

19. *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.

20. *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.
21. Do not use XA transactions unless distributed transaction functionality is required by your application. Distributed transactions are much slower than standard transactions (sometimes by as much as a factor of 500 to 1).

22. *Set the `com.solarmetric.kodo.TransactionCacheClass` property to `com.solarmetric.kodo.runtime.ClassGroupStateManagerSet`: See the `StateManagerSet` documentation for details.*

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](#)

of JDO and Java. For more information on these subjects, see the following URLs:

This tutorial also assumes some familiarity with the concepts covered in the Kodo JDO Reference Guide.

1.1. Tutorial Requirements

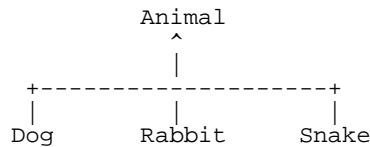
This tutorial requires that JDK 1.2 or greater be installed on your computer, and that 'java' and 'javac' are in your path when you open a command shell. See README.txt for more information on requirements and installation procedures.

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



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`

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 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: Note the use of

- **`java tutorial.AnimalMaintenance details Snake "giTract.contains (animal) && animal.name == \'Roger\'"`**

one of the snakes ate it. In order to determine who ate Roger Rabbit, you could run a query like this: 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. Kodo J2EE Tutorial

Introduction to the J2EE Tutorial

By deploying Kodo into a J2EE environment, one can maintain the simplicity and performance of Kodo JDO, while leveraging J2EE technologies such as container managed transactions (JTA/JTS), enterprise objects with remote-invocation (EJB), and managed deployment of multi-tiered applications via an application server. This tutorial will demonstrate how to deploy Kodo-based J2EE applications as well as showing some basic design techniques for this combination. The sample application attempts to model a basic garage catalog system. While the application is relatively trivial, the code has been constructed as to illustrate simple patterns and solutions to common problems when using Kodo JDO in the enterprise environment.

1.1. Prerequisites for the Kodo J2EE Tutorial

This tutorial assumes that you have installed Kodo and setup your classpath according to the installation instructions appropriate for your platform. In addition, this tutorial requires that you have installed and configured a J2EE-compliant application server, such as JBoss, WebSphere, WebLogic, Borland Enterprise Server, or SunONE. This tutorial may be adaptable to your application server with small changes; refer to your application server's documentation for any specific classpath and deployment descriptor requirements.

This tutorial assumes a reasonable level of experience with Kodo and JDO. We provide a number of other tutorials for basic Kodo and JDO concepts, including enhancement, schema mapping, and configuration. This tutorial also assumes a basic level of experience with J2EE components, including EJB, JNDI, JSP, and EAR/WAR/JAR packaging. Sun and/or your application server company may provide tutorials to get familiar with these components.

This tutorial also requires some J2EE libraries to be in your classpath to compile the EJBs. The EJB libraries are downloadable from Sun and is probably distributed with your application server.

In addition, this tutorial uses Ant to build the deployment archives. While this is the preferred way of building a deployment of the tutorial, one can easily build the appropriate JAR, WAR, and EAR files by hand, although that is outside the scope of this document.

Chapter 1. J2EE Installation Types

Every application server has a different installation process for installing J2EE components. Kodo can be installed in a number of ways which may or may not be appropriate to your application server. While this document focuses mainly upon using Kodo as a JCA resource, there are other ways to use Kodo in the J2EE environment.

- **JCA:** Kodo implements the JCA 1.0 spec, and the `kodo.rar` file that comes in the JCA directory of the distribution can be installed as any other JCA connection resource. This is the preferred way to integrate Kodo into the J2EE environment. It allows for simple installation (usually involving uploading or copying `kodo.rar` into the application server), guided configuration on many appservers, as well as dynamic reloading for upgrading Kodo to a newer version. The drawback is that older application servers have flaky or non-existent JCA support as the spec is relatively new.
- **EEPersistenceManagerFactory:** This remains the most compatible way to integrate Kodo into the J2EE environment, though this is not seamless and can require a fair bit of custom application server code to manage manually binding an instance of `EEPersistenceManagerFactory` into the JNDI tree. This is somewhat offset by avoiding JCA configuration issues as well as being able to tailor the binding and start-up process.
- **JDBCPersistenceManagerFactory:** Nothing prevents an application from using `com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory` similar to standalone applications. `JDBCPersistenceManagerFactory` can be bound into JNDI just as `EEPersistenceManagerFactory`. However, you lose the ability to integrate the JDO transaction into the J2EE container's transaction.

Chapter 2. Installing Kodo JCA

2.1. JBoss 3.0

Included in the JCA directory of the distribution is `kodo-service.xml`. This file should be edited to reflect your configuration, most notably connection and license key values. Enter a JNDI name for Kodo to bind to (the default in `kodo-service.xml` is `kodo`). Stop JBoss. Copy `kodo.rar` and `kodo-service.xml` to the deploy directory of your JBoss server installation (e.g. `jboss-3.0.6/server/default/deploy`). In addition, you should also place the appropriate JDBC driver jar in the lib directory of your JBoss installation (i.e. `jboss-3.0.6/lib`).

To verify your installation, watch the console output for exceptions. In addition, you can check to see if Kodo was bound to JNDI. Open up your jmx-console (<http://yourhost:yourport/jmx-console>) and select the JNDIView service. If Kodo was installed correctly, you should see `com.solarmetric.kodo.impl.jdbc.JDOConnectionFactory` bound at the JNDI name you specified. You have now installed Kodo JCA.

2.2. JBoss 3.2

Installing in JBoss 3.2 is very similar to JBoss 3.0. Instead of editing and deploying `kodo-service.xml`, configuration is controlled by `kodo-ds.xml`, also in the JCA directory. Again, configuration involves supplying a JNDI name to bind Kodo, and setting up configuration values. These values are simple XML elements with the configuration property name as element name. This `kodo-ds.xml` and `kodo.rar` should be deployed to the deploy directory of JBoss. The installation is otherwise the same as JBoss 3.0.

2.3. WebLogic Versions 6.2 to 7.x

Installation of Kodo into WebLogic requires 3 steps. First ensure that the appropriate JDBC driver is in the system classpath. In WebLogic 6.2.x, this should be in the `startWebLogic.sh/cmd` in your domain directory (`$WL_HOME/config/mydomain`). In WebLogic 7, this file is the `startWLS.sh/cmd` file in the `$WL_HOME/server/bin` directory. In addition, the JDO base jar (`jdo1_0.jar`) should be also added to the classpath so that your JDO classes can be loaded. While this file can be placed inside an ear file, putting it in the system classpath will reduce class resolution conflicts.

The `kodo.rar` file should then either be copied to the applications directory of your domain, or uploaded through the web admin interface. To upload using the web admin console, by selecting `mydomain/Deployments/Connectors` in the left navigation bar and selecting "Install a new Connector Component." Browse to `kodo.rar` and upload it to the server.

You should see `kodo` listed now in the Connectors folder in the navigation pane. Select it and select `Edit Connector Descriptor`. Under RA, expand `Config Properties` in the left pane and enter the appropriate values for each property. *Be sure to select Apply for every property.* In addition, you should provide a JNDI name for Kodo by selecting `Weblogic RA` from the navigation panel, entering an appropriate JNDI name, and selecting `Apply`. When you are done configuring Kodo, select the root (`kodo.rar`) of the navigation pane. Select `Persist` to save your configuration properties and save it to the active domain configuration.

You should see WebLogic attempt to deploy Kodo in the system console. When it is done, you should return to the main admin web console. Ensure that Kodo is deployed to your server by selecting `Targets` and adding your server to the chosen area. Kodo should now be deployed to your application server.

To verify the installation, you can view the JNDI tree for your server. Select the server from the admin navigation panel, right click on it, and select `View JNDI Tree` from the context menu. You should now see Kodo at the JNDI name you provided. You have now installed Kodo JCA.

2.4. WebLogic 8.1

Note

In its current version (8.1.0), there are a number of issues with classloaders in WebLogic's handling of EAR and RAR files. These instructions are aggressive in resolving potential issues which may no longer be issues in later releases of WebLogic.

First, ensure that your JDBC driver is in your system classpath. In addition, you will be adding `jdo1_0.jar` to the system classpath. You can accomplish this by editing `startWebLogic.sh/.cmd`.

The next step is to deploy `kodo.rar` from the JCA directory of your Kodo installation. Create a directory named `kodo.rar` in the `applications` directory of your domain. Un-jar `kodo.rar` into this new directory (without copying `kodo.rar` itself). Then extract `kodo-jdo-runtime.jar` in place and remove the file:

```
applications> mkdir kodo.rar
applications> cd kodo.rar
kodo.rar> jar -xvf /path/to/kodo.rar
kodo.rar> jar -xvf kodo-jdo-runtime.jar
kodo.rar> rm kodo-jdo-runtime.jar
```

Now you should configure Kodo JCA by editing `META-INF/ra.xml` substituting `config-property-value` stanzas with your own values. You can comment out properties (`config-property` stanzas) which you are not using or leaving at default settings. Edit `META-INF/weblogic-ra.xml` to configure the JNDI location to which you want Kodo to be bound.

Now you can start WebLogic and use the console to deploy Kodo JCA. Browse to your WebLogic admin port (<http://yourhost:7001/console>) and browse to the `Connectors` (`Deployments -> Connector Modules`) section. If you have autodeploy on and you have correctly installed Kodo JCA, you should see Kodo listed in the connectors.

If Kodo has not been deployed, select `Deploy a new Connector`. Browse to and select `kodo.rar` and select `Target Modules` to ensure that Kodo is accessible to the proper servers.

If you have installed Kodo correctly, at this point, one should be able to see Kodo bound to the JNDI location which you specified earlier.

2.5. WebSphere 5

Note

We have seen some problems with `rmic` in WebSphere. We have included a patch jar that is a temporary solution until WebSphere fixes the way `ResourceBundles` are loaded during `rmic`. Copy `i18nhelper_websphere_patch.jar` to the `lib` directory of your WebSphere installation if you see exceptions deploying JDO-based EJBs. These exceptions are usually of the `ResourceNotFoundException` type.

Installation is best performed by using the web admin interface. Open the admin console either by the `Start` menu item (in Windows), or by manually navigating to the admin port and URL appropriate to your installation. Select `Resources / Resource Adapters` from the left navigation panel. Select `Install Rar` on the list page. On the following screen upload `kodo.rar` to the server. On the `New` page, enter a name for the new Kodo installation such as `Kodo JCA` and select `Ok`.

You should now be back to the `Resource Adapters` list page. Select the name of the Kodo installation you provided. Click on the link marked `J2C Connection Factories`. This is where you can configure a particular instance of Kodo's JCA implementation. Select `New` and you will be brought to a configuration page. *Be sure to fill in property values for Name and JNDI Name*. Select `Apply`.

After the page refreshes, select the `Custom Properties` link at the bottom of the page. On the `Custom Properties` page, you can enter in your connection and other configuration properties as necessary.

When you are done providing configuration values, you will want to save your changes back to the system configuration. Select the `Save` link on the page or the `Save` link in the menu bar. You have now installed Kodo JCA.

2.6. SunONE Application Server

Installation in SunONE application server requires first providing JDO the proper permissions. This is accomplished by editing the `config/server.policy` for the server you are dealing with. Edit the file to include the following line into a `grant { }` stanza. and restarting SunONE.

```
permission javax.jdo.spi.JDOPermission "*";  
line into a grant { } stanza. and restarting SunONE.
```

Deploying the RAR file requires adding a SunONE specific deployment file to the `kodo.rar` file. Take the `sun-ra.xml` file provided in the JCA directory of your Kodo installation. Edit the `jndi-name` attribute to the JNDI name of your choice. Add `<property>` elements that correspond to the `<config-property-name>` in `ra.xml` to configure connection info and other configuration elements. Now update `kodo.rar` to include `sun-ra.xml` to the `META-INF` virtual directory in the archive:

```
mkdir META-INF  
copy sun-ra.xml META-INF  
jar -uvf kodo.rar META-INF/sun-ra.xml
```

Browse to the web admin console of SunONE in your browser. Select your server under `App Server Instances` and expand to `Applications/ Connector Modules`. Select `Deploy` and upload your new `kodo.rar` file. Enter an application name, and click the `Select` button. Apply your changes by selecting the link in the top right.

Note

Unfortunately, while this should be the recommended instructions, SunONE Application Server may not observe its own configuration file format. This will evidence itself with exceptions as if your configuration values had not been set at all. To bypass this problem, extract `kodo.rar` into a temporary directory. Edit `ra.xml` and provide your configuration values directly into the `<config-property-value>` elements into the proper `<config-property>` stanzas.

If you have installed Kodo correctly, you should see `kodo` listed in the `Connector Module`. You have now installed Kodo JCA.

2.7. Borland Enterprise Server 5.2

Note

While we support BES using the "Lite Transaction Manager" which should be more than adequate for most enterprise applications, Borland has yet to release a JTA interface into their "2PC Transaction Manager." While Kodo supports XA datasources, integrating into container managed transactions using the 2PC Transaction Manager may require either Borland support or custom coding to wrap their OTS implementation into a J2EE-compliant interface. See the documentation regarding `ManagedRuntimeClass` to see how to notify Kodo of the 2PC Transaction Manager wrapper. Note that the 2PC Transaction Manager can be quite slow in comparison to the Lite Transaction Manager.

Due to classloader issues, as well as configuration of the RAR itself, some basic expertise in `jar` and `unjarring` is required to install Kodo into BES. Otherwise, we recommend deploying Kodo, then switching to single classpath for the entire system through the Admin tool (which prevents hot deploy) to ease classpath conflict issues. First extract `kodo.rar` to a temporary directory:

```
mkdir tmp
cd tmp
jar -xvf ../kodo.rar
```

From there, remove/move some jars that will cause class conflict issues with either your application or BES. Remove `jakarta-commons-logging-*.jar`. Move (and delete from the temporary directory) `jdo1_0.jar` to the system classpath (e.g. `var/servers/your_server/partitions/lib/system`).

You must configure the RAR by editing the `ra-borland.xml` file included in the JCA directory of your Kodo installation. Move this file into the `META-INF` directory of the expanded RAR file. Refer to the DTD and Borland's documentation on advanced features of the deployment descriptor, including deployment and security options.

With this completed, you can re-jar the expanded contents into a new `.rar` file to be deployed using `iastool` or the console interface. Before deploying, first *enable Visiconnect* on the partition using the console or the command-line utilities. This will activate JCA support in the application server. Then restart BES so that Visiconnect can activate and so that `jdo1_0.jar` is added to the runtime classpath. When deploying, stubs and verification do not need to be processed on the file and may simplify the deployment process.

```
tmp> jar -cvf kodo-bes.rar *
```

After a restart, Kodo should now be deployed to BES. You should be able to find Kodo at the JNDI location of `serial://kodo` in your application as well as be visible in the JNDI viewer in the console. You have now installed Kodo JCA.

Chapter 3. Installing the J2EE Sample Application

Installing the sample application involves first compiling and building a deployment archive (.ear) file. This file then needs to be deployed into your application server.

3.1. Compiling and Building The Sample Application

Navigate to the `samples/j2ee` directory of your Kodo installation. Compile the source files in place both in this base directory as well as the `ejb` directory:

```
javac *.java  ejb/*.java
```

Enhance the `Car` class.

```
jdoc -properties /path/to/kodo.properties package.jdo
```

Run SchemaTool using a `kodo.properties` with connection info (e.g. Driver, URL, etc.) corresponding to your JCA installation:

```
schematool -properties /path/to/kodo.properties  
-action refresh package.jdo
```

Configure options in `samples.properties` to match your JCA installation, most notably the JNDI name to which you have bound Kodo (it defaults to `kodo`).

Warning

This step (editing `samples.properties`) is *very important* as this value can be quite different for each appserver and each configuration.

Be sure that the setting you put for `pmf.jndi` matches not only your configured setting but also what your application server may prefix the configured name with.

On JBoss, for example, JBoss will prefix the JNDI name with `java:/` and Borland Enterprise Server looks at the `serial://` context. Refer to your JNDI tree and the documentation for your application server for further details.

Build an J2EE application archive by running Ant against the `build.xml`. This will create `samplej2ee.ear`. This ear can now be deployed to your appserver.

```
ant -f build.xml
```

3.2. Deploying Sample To JBoss

Simply place the ear file in the `deploy` directory of your JBoss installation. You can use the above hints to view the JNDI tree to see if `samples.j2ee.ejb.CarHome` was deployed to the JNDI name of the classname.

3.3. Deploying Sample To WebLogic 6.2 to 7.x

Simply place the ear file in the `applications` directory of your WebLogic domain. Production mode (see your

startWebLogic.sh/cmd file) should be set to false to enable auto-deployment. If the application was installed correctly, you should see `sample-ejb` listed in the Deployments/EJB section of the admin console. In addition you should find `CarHome` listed in the JNDI tree under `myserver->samples->j2ee->ejb` .

3.4. Deploying Sample To WebLogic 8.1

Create a new directory named `samplej2ee.ear` in the `applications` directory of your WebLogic domain. Extract the EAR file (without copying the EAR file) to this new directory:

```
applications> mkdir samplej2ee.ear
applications> cd samplej2ee.ear
samplej2ee.ear> jar -xvf /path/to/samplej2ee.ear
```

Deploy the application by using the admin console (Deployments -> Applications). If you have autodeploy on, you may see the application already deployed. If you do not, select `Deploy a new Application`. Select `samplej2ee.ear` and deploy to the proper server targets. If you have installed the application correctly, you should find `CarHome` listed in the JNDI tree under `myserver->samples->j2ee->ejb` .

3.5. Deploying Sample To SunONE

Browse to the admin console in your web browser. Select `Applications / Enterprise Applications` from the left navigation panel. Select `Deploy...` and in the following screen upload the `samplej2ee.ear` file to the server. Apply your changes by selecting the link in the upper right portion of the page. You should now see `samplej2ee` listed in the `Enterprise Applications` folder of the navigation panel.

3.6. Deploying Sample To WebSphere

Browse to the admin console in your web browser. Select `Applications / Install New Application` from the left navigation panel. Select the path to your `samplej2ee.ear` file and press `Next`.

On the following screen, leave the options at the default and select `Next`. On the following screen (`Install New Application->Step 1`), ensure that the `Deploy EJBs` option is checked. Leave other options at their defaults.

Move on to `Step 2`. On this screen enter `"samples.j2ee.ejb.CarHome"` as the JNDI name for the Car EJB. Continue through the remaining steps leaving options at the defaults. Select `Finish` and ensure that the application is deployed correctly.

Save the changes to the domain configuration by either selecting the `Save` link that appears after the installation is complete or by selecting `Save` from the top menu.

To verify your installation, select `Applications / Enterprise Applications` from the left navigation panel. `Sample-KodoJ2EE` should be listed in the list. If the application has not started already, select the checkbox next to `Sample-KodoJ2EE` and select `Start`.

3.7. Deploying Sample To Borland Enterprise Server 5.2

Note

First be sure to edit `samples.properties` to change the JNDI location (to `serial://kodo`). In addition, serial JNDI instances cannot be treated as remote objects, so be sure to disable JNDI narrowing by uncommenting the `pmf.narrow` property and setting to `false`

Deploy the EAR file using `iastool` or the console. Be sure that you have followed the JCA instructions for BES as well as editing the `samples.properties` as noted above. You should be able to see the `CarEJB` located in JNDI located at the home classname.

Chapter 4. Using The Sample Application

The sample application installs itself into the web layer at the context root of sample. By browsing to `http://yourserver:yourport/sample`, you should be presented with a simple list page with no Cars. You can edit, add, delete Car instances. In addition, you can query on the underlying Car PersistenceCapable instance by passing in a JDOQL query into the marked form (such as `model=="Some Model"`).

Chapter 5. Sample Architecture

The application is a simple enterprise application that demonstrates some of the basic concepts necessary when using Kodo in the enterprise layer.

The core model wraps a Stateless SessionBean facade around a PersistenceCapable instance. Using a SessionBean provides both a remote interface for various clients as well as providing a transactional context in which to work (and thus avoiding any explicit transactional code).

This SessionBean uses the Data Transfer Object pattern to provide the primary communication between the application server and the (remote) client. The Car instance will be used as the primary object upon which the client will work.

This model can be easily adapted to using EntityBeans. See our sample ejb directory and JDOEntityBean for more details on how to using JDO to power your EntityBean.

- `samples/j2ee/Car.java`: The core of the sample application. This is the PersistenceCapable class that Kodo will use to persist the application data. Instances of this class will also fill the role of data transfer object (DTO) for EJB clients. To accomplish this need, Car implements `java.io.Serializable` so that remote clients can access these as parameters and return values from the EJB.
- `samples/j2ee/package.jdo`: The JDO metadata file for the package. Note that some appservers have a problem with the way we include an internal DTD. If you see a lot of illegal/malformed character exceptions, uncomment out the DTD declaration in this file.
- `samples/j2ee/SampleUtilities.java`: This is a simple facade to aggregate some core JDO functionality into some static methods. By placing all of the functionality into a single facade class, we can reduce code maintenance, as well as having the added advantage of being able to access Kodo JDO from other portions of the application such as a servlet or JSP (though this functionality is not demonstrated in this sample). In addition, some simple utility functions such as JNDI helper methods are also in this class.
- `samples/j2ee/ejb/Car*.java`: The source for the CarEJB Session Bean. Clients can use the CarHome and CarRemote interfaces to find, manipulate, and persist changes to Car transfer object instances. By using J2EE transactional features, the implementation code in CarBean.java can be focused almost entirely upon business and persistence logic without worrying about transactions.
- `samples/j2ee/jsp/*.jsp`: The web presentation client. They are JDO agnostic, and use the CarEJB Session Bean and the Car transfer object to do all the work.
- `samples/j2ee/resources/*`: Files required to deploy to the various appservers, including J2EE deployment descriptors, WAR/ EJB/ EAR descriptors, as well as appserver specific files.
- `samples/j2ee/build.xml`: A simple Ant build file to help in creating a J2EE EAR file.
- `samples/j2ee/samples.properties`: A simple .properties file to tailor the sample application to your particular appserver and installation

Chapter 6. Code Notes and J2EE Tips

1. JDO PersistenceCapable instances make for top candidates for using the Data Transfer Object Pattern. This pattern attempts to reduce network load, as well as group business logic into concise units. For example, CarBean.edit () allows one to ensure that all values are correct before committing a transaction, instead of sequentially calling getters and setters on the session facade. This is especially true when using RMI such as from a Swing based application connecting to an application server.

CarEJB works as a Session Bean facade, to demarcate transactions, provide finder methods, and to encapsulate complex business logic at the server level.

2. PersistenceCapable instances using datastore identity lose all identity upon serialization (which happens when it is returned from an EJB method). While there are numerous ways to solve this problem, the sample uses the clientId field of Car to store the String version of the datastore id. *Note that this field is not persistent.* This field ensures that the client and EJB always share the same identity for a given Car instance.

Note

Note that this situation is slightly different for application-identity. While PersistenceCapable instances under application-identity still lose their id instance, recreating it from the server side should be trivial as the fields on which it is based are still available.

Other ways of solving this problem include:

- Transmitting the object id first or otherwise storing identity with the client:

```
Object oid = dogRemote.findByQuery ("// some query");
Dog dog = dogRemote.getDogForOid (oid);
// changes happen to dog
dogRemote.save (oid, dog);
Transmitting the object id first or otherwise storing identity with the client:
```

- Wrapper objects that store the PersistenceCapable instance, object id instance, as well as potentially other application specific data:

```
public class DogTransferObject
{
    private Dog dog;
    private Object oid;
    private String authentication; // some application specific client data
}
application specific data:
```

Other ways of solving this problem include:

3. PersistenceManager.close () should be called at the end of every EJB method. In addition to ensuring that your code will not attempt to access a closed PersistenceManager, it allows the JDO implementation to close up resources no longer needed. Since a PersistenceManager is created for every transaction, this can increase the scalability of your application.
4. You should not use PersistenceManager.currentTransaction () or any other javax.jdo.Transaction methods. Instead, use the JTA transactional API instead. SampleUtilities includes a helper method to obtain a UserTransaction instance from JNDI (see your application server's documentation on the proper JNDI lookup).

5. While serialization of PC instances is relatively straight forward, there are several things to keep in mind:
- Collections and Iterators returned from Query instances become closed and inaccessible upon method close. If the client is receiving the results of a query, such as in `CarBean.query ()`, the results should be transferred to another fresh Collection instance.
 - While "default-fetch-group" values will always be returned to the client upon serialization, lazily loaded fields will not as the PersistenceManager will have been closed before those fields attempt to serialize. One can either simply access those fields before serialization, or one can use the utility methods `retrieve ()` and `retrieveAll ()` methods in PersistenceManager. Note that these methods are not recursive. If one needs to go through multiple relations, `retrieve ()` needs to be called at each relational depth. This is an intentional limitation in the specification to prevent the entire object graph from being serialized and/or retrieved.

While serialization of PC instances is relatively straight forward, there are several things to keep in mind:

6. It is not necessarily required that one uses EJBs and container-managed transactions to demarcate transactions, although that is probably the most common. By obtaining a `javax.transaction.UserTransaction`, one can finely control transactions in a bean-managed EJB. Furthermore, one could access the JDO layer using the UserTransaction API free of EJBs altogether, bypassing the JDO transaction API. While not preferred, this technique could prove useful in tailoring small portions of your application outside the EJB context for finer transactional control.
7. We ship compiled and included in our source some convenient base classes that encapsulate much of the code that is laid out in this example for clarity. `JDOBean`, `JDOSessionBean` and `JDOEntityBean` include most of the functionality of `SampleUtilities`, as well as handling common EJB interface implementations such as `setEntityContext ()`. To use these classes, it is recommend that you put Kodo's jars into the system classpath and not into the ear. This can cause classloader problems due to the multiple locations that these classes could be loaded from.
8. PersistenceManagers are allocation on a per-Transaction basis. Calling `getPersistenceManager ()` from the same PersistenceManagerFactory within the same EJB method call will always return the same instance.

```
SampleUtilities.getPersistenceManager ()  
== SampleUtilities.getPersistenceManager (); // will always be true  
PersistenceManagerFactory within the same EJB method call will always return the same instance.
```

Part VI. 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](#)

on these subjects, see the following URLs:

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.txt. 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 `reversetutorial_database.properties`, `reversetutorial_database.script`, and `reversetutorial_database.data`: These 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. Persistence manager factories 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-supplied 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`: Properties file containing Kodo-specific and standard JDO configuration settings.

For this tutorial, make sure the following properties are set to the given values:

```
javax.jdo.option.ConnectionDriverName=org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionUserName=sa
javax.jdo.option.ConnectionPassword=
javax.jdo.option.ConnectionURL=jdbc:hsqldb:reversetutorial_database
```

For this tutorial, make sure the following properties are set to the given values:

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.

The tutorial uses the following files:

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**.
- **jdgc** 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.

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 schema generator, which 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
```

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. For the curious, this XML format is documented [here](#).
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
```

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, and a `reversetutorial.mapping` file.

5. The `.mapping` file generated by the reverse mapping tool is in an XML format used in Kodo's R&D codebase. In order to use the mapping information in current Kodo versions, we have to *import* it. To import mapping data, we use the aptly-named import tool. 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 reversetutorial.mapping
to compile the generated classes before the import:
```

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 `reversetutorial.jdo` metadata file. It contains the necessary standard JDO metadata, plus, thanks to our use of the import tool, 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 reversetutorial.jdo
```

This step runs the Kodo JDO enhancer on the `reversetutorial.jdo` file mentioned above. The `reversetutorial.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.

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 is discussed in the JDO Overview . 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 that 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.

Note

For a complete treatment of the reverse mapping tool, including how to customize generated classes and mappings, see the Reference Guide.

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.
- `commons-collections.jar`: Used by the Kodo Reverse Schema tool only. See <http://jakarta.apache.org/commons/collections.html>
- `commons-lang.jar`: Used by the Kodo Reverse Schema tool only. See <http://jakarta.apache.org/commons/lang.html>
- `commons-pool.jar`: Used by the Kodo Reverse Schema tool only. See <http://jakarta.apache.org/commons/pool.html>
- `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-1.0.3.jar`: Logging wrapper classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/logging.html>
- `jakarta-regexp-1.1.jar`: Regular expression libraries. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/regexp/>
- `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-hsqldb-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`

All JDO options are supported, with the following exceptions:

- 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.

features specific to relational data stores:

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`.

Following is a table of the java class to SQL type that is generated by the `schematool`.

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

Following is a table of the java class to SQL type that is generated by the `schematool`.

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)

Following is a table of the java class to SQL type that is generated by the `schematool`.

Table D.3. SQL Type Mappings III

	SHORT	DATE	BLOB	BIGINTEGER	BIGDECIMAL
MySQL	SMALLINT	DATETIME	BLOB	DECIMAL(200)	DECIMAL(200,8)

Appendix D. SQL Types

	SHORT	DATE	BLOB	BIGINTEGER	BIGDECIMAL
					00)
Hypersonic SQL	SMALLINT	TIMESTAMP	VARBINARY	DECIMAL	DECIMAL
DB2	SMALLINT	TIMESTAMP	BLOB (1M)	BIGINT	DOUBLE
Generic	SMALLINT	TIMESTAMP	BLOB	BIGINT	DECIMAL
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

2.5.8 -- 11 January 2004

- Bugfixes
 - Fixed bug with improper lookup of unloaded related objects from cache (bug 836).
 - Fixed intermediate value caching issue with data cache lookups, improving performance when loading graphs of cached values.
- Notable changes
 - Improved connection pool warm-up speed by reducing concurrency when a connection is requested from the pool while no connections are available and the pool is not full.

2.5.7 -- 23 December 2003

- Bugfixes
 - Fixed bug with improper caching of in-memory query results (bug 822).
 - Data cache now properly deals with loading of the default fetch group as soon as an object is initialized from the cache (bug 819).
 - Queries executed after modifications have occurred in a transaction are no longer stored in the query cache, in case the transaction is later rolled back and to provide appropriate transactional isolation (bug 820).

2.5.6 -- 8 December 2003

- Bugfixes
 - Datastore cache now caches OID of one-to-one related objects with owning object.
 - Fixed problem where an object that is removed and then re-added to a relation in the same transaction might not be re-added properly if `FlushBeforeQueries` is true and `IgnoreCache` is false.

2.5.5 -- 18 October 2003

- Bugfixes

- Fixed synchronization problem in `EEFactoryHelper` that could result in a `ConcurrentModificationException` when Kodo is used under high load with managed transactions.
- Fixed problem with checking the optimistic lock version of a subclass that uses a vertical inheritance mapping strategy.
- Notable changes
 - Refactored data caching to not lock the cache as a whole when loading data from it or storing data into it. This change improves the concurrency of the cache.
 - Removed the `com.solarmetric.kodo.DataCacheConnects` property, as it is not needed now that locks are not obtained on the data cache.
 - Made assorted minor tweaks to prepared statement and query caching.

2.5.4 -- 7 October 2003

- Bugfixes
 - Fixed bug with extents not closing resources with certain `ResultList` implementations due to internal iterators not closing.
 - Fixed bug with data caching and incremental flushing and loading that could result in incorrect `OptimisticLockExceptions` being thrown.
 - Fixed bug with data caching that could result in deadlocks when used in conjunction with table-level or page-level locking.
- Notable changes
 - Added the `com.solarmetric.kodo.DataCacheConnects` property to determine whether the data cache obtains a connection before each cache access. Note that this property defaults to `false`, which mirrors Kodo 2.5.2 behavior. Users who experienced data cache hangs in Kodo 2.5.2 because of empty connection pools should set this property to `true` to mirror Kodo 2.5.3 behavior.

2.5.3 -- 27 August 2003

- Bugfixes
 - Fixed bug with invalid `SQLServer SQL92` generation when using pessimistic locking.
 - Addressed performance issues caused by recomputing persistent type lists and subclass lists too often.

- Fixed result list implementation used by the query caching framework to properly lazily load results.
- Fixed DataCacheStoreManager to properly deal with creating a new query based on a template that is a CacheAwareQuery, and changed CacheAwareQuery to have a writeReplace() method that returns the delegate query object rather than the cache-aware query.
- Fixed bug that prevented custom query extensions from being recognized.
- Fixed bug with data caching and incremental flushing that could result in incorrect OptimisticLockExceptions being thrown.
- Changed on-demand ConnectionRetainMode to only obtain a single connection per PersistenceManager. In other words, if a PM is using a connection (for example, while iterating a large query result), and it performs an operation that requires a connection, it will use the previously-obtained connection rather than obtaining a new connection. This reduces resource consumption, and helps to avoid possible race conditions while obtaining connections. If the old on-demand ConnectionRetainMode is necessary for some reason, it can be activated by setting `com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode` to `legacy-on-demand`.
- Fixed potential race condition when performing operations on the data cache that might require a trip to the data store.
- Notable changes
 - Improved validation of application ID object-id classes may cause errors when enhancing or deploying malformed classes. These should be easily fixable by modifying your object-id classes to conform to the JDO specification rules.
 - Changed OnDemandForwardResultList to not use weak or soft references, but instead to optionally use a scrolling window to prevent memory growth as large result sets are iterated.

2.5.2 -- 4 July 2003

- Bugfixes
 - The repackaged concurrent.util APIs have been included in the released jars.
 - Fixed potential rounding bug where the fractional parts of a Date field can be doubled when using JDK 1.4.1.
 - Fixed problem where AutoIncrementSequenceFactory was not working for SQL Server.
- Notable changes
 - Changed the ConnectionRetainMode fix that was made in 2.5.1 to not actually close the PersistenceManager, replicating the behavior of 2.5.0 and earlier. This means that session beans that return live JDO objects without detaching them or copying them into data transfer objects will continue to function as with 2.5.0 and earlier releases. It is likely that Kodo 3.0 will deal with this differently, possibly including a mode to allow the current, more lenient behavior.

2.5.1 -- 4 July 2003

- New Features
 - Added ability to use database-specific outer join syntax. Coded Oracle 8i outer joins into Oracle dictionary.
 - Borland Enterprise Server is now supported, meaning that the AutomaticManagedRuntime class knows about where Borland puts its transaction manager in JNDI. In addition, the J2EE tutorial has been updated with detailed deployment instructions for Kodo as a JCA Resource Adapter.
- Bugfixes
 - Eclipse/WSAD plugin ClassLoader problems resolved. Please update the plugins/com.solar.../kodo-jdo.jar to the latest release.
 - Fixed ConnectionRetainMode=persistence-manager to correctly close resources when used in a container-managed transaction context. This fix may cause applications that use session beans but do not properly (serialize | clone | makeTransient) persistence-capable objects returned from the session beans to throw exceptions stating that a PersistenceManager has been closed. This can only be an issue if your session bean is deployed to the same JVM as the EJB client code.
 - Deadlocking problem with upgrading read locks in data cache has been resolved.
 - Optimistic lock version problem when RetainValues is false was resolved.
 - Connection leak problem in AutoIncrementSequenceFactory was resolved.
 - Improved performance of JDO class initialization in environments with potentially slow classloaders, such as JBoss.
- Notable changes
 - The included distribution of Apache Commons Logging is now 1.0.3. Be sure to update your classpath accordingly as there were some difficult to diagnose configuration bugs in 1.0.2. The JCA rar file has been updated with the newer version.
 - The UsePreparedStatements option has been removed: prepared statements are now always used for all drivers.
 - Made all queries using unbound variables use SELECT DISTINCT.
 - Kodo once again forces the prepared statement pool size to zero when using Microsoft's JDBC driver. You can prevent Kodo from doing this by setting the com.microsoft.jdbc.sqlserver.SQLServerDriver.nopool system property. See http://bugzilla.solarmetric.com/show_bug.cgi?id=501 for details.

2.5.0 -- 5 June 2003

- New features

- Custom fetch groups are now supported. See the fetch group documentation and the FetchGroups configuration documentation for more information.
- Participation in a global XA-compliant transaction is now possible in a managed environment.
- Multi-table mappings now permit different tables to have different primary key column names.
- The `ProxyManager` now includes capabilities to proxy user-defined mutable field types that are not part of the JDO specification.
- Kodo now supports auto-increment columns when using datastore identity. See the `sequence-factory-class` metadata extension and `SequenceFactoryClass` configuration property documentation for usage details.
- New flush API allows the modifications made in a transaction to be incrementally flushed to the database before transaction commit time. See the `com.solarmetric.kodo.runtime.KodoTransaction.flush()` JavaDoc for details.
- Added subclasses of `JDOUserException` for special cases that are of interest:
`com.solarmetric.kodo.runtime.OptimisticLockException` and
`com.solarmetric.kodo.runtime.ObjectNotFoundException`.
- New Kodo J2EE integration tutorial. See the documentation as well as the source code before proceeding. Currently, the tutorial includes instructions for WebLogic 6.2 and higher, SunONE Application Server 7, WebSphere 5, and JBoss 3.x.
- The association between a `PersistenceManager` and a `JDBC Connection` can now be configured. The default behavior is the same as in earlier versions of Kodo -- connections are obtained on-demand. Additionally, Kodo can be configured to retain a connection for the duration of a transaction (both optimistic and pessimistic) or for the duration of a `PersistenceManager`'s life cycle. This behavior is controlled with the `com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode` configuration property.
- Enhancement-time validation of JDO metadata has been improved. This may result in errors next time you recompile and re-enhance your persistence-capable classes.
- Modified persistence-capable classes can be grouped by class before being flushed to the data store, increasing the potential for performance benefits due to statement batching. See the `ClassGroupStateManagerSet` documentation for details about this option.
- Added direct support for custom collections and maps that implement `ProxyCollection` or `ProxyMap`, and for fields that implement `Proxy`.
- Informix IDS is now a supported database.
- Second-class objects that are externalizable to Strings can now be stored to string fields. See the `Storing Second Class Objects via Stringification` documentation for more information.
- Data caching framework now caches JDOQL queries. See the Kodo JDO Query Caching section for more details.
- Data caching framework includes semantics for specifying a timeout for a given class. See the Metadata documentation for more details.
- Different classes can use different `PersistenceManagerFactory` caches, allowing for varying cache policies on a per-class basis.
- A transaction event listener framework has been created. This framework allows listeners to be notified of transaction begin, commit, and rollback events on a per-`PersistenceManager` level, and of transaction

commits on a per-PersistenceManagerFactory level. Additionally, this framework allows transaction commit notification to be propagated to remote PersistenceManagerFactory objects. See event notification framework documentation for more information.

- The schema manipulation done by the SequenceFactory to initialize any database-specific tables to store sequence information is now done when the schematool is run, rather than at runtime.
- Queries have received a major overhaul. Queries now support unbound variables, Collections as parameters to generate SQL IN (...) clauses, traversing fields of persistence-capable parameters, and more. The SQL produced by queries is also much more efficient.
- The Query FilterListener API has changed, and the default set of available FilterListeners has been enhanced with a few new and powerful extensions. Some of the old extensions have been deprecated, so check the Query Extensions section of the documentation for details on the new extensions framework. Additionally, it is unlikely that existing custom query extensions will continue to work.
- Added the `com.solarmetric.kodo.impl.jdbc.UseSQL92Joins` configuration property. Set this property to `true` to use SQL 92-style joins in queries, including left outer joins where appropriate. (This is the default value.) You can also set this property on an individual query instance; see the `com.solarmetric.kodo.impl.jdbc.query.JDBCQuery` class Javadoc for details.
- `PersistenceManager.newQuery(Class)` and `PersistenceManager.getExtent(Class)` can now take an interface as a parameter, even when multiple separate inheritance hierarchies implement the interface and exist in the data store. Ordering for queries will work as expected, but it should be noted that an ordered query that is executed against multiple tables will result in partial loss of large result set support, such that attempting to access element N in the Collection returned from `Query.execute()` will force the results 0..N-1 to be instantiated so that an in-memory comparison of the homonegous objects can take place.
- The new properties `com.solarmetric.kodo.ResultListClass` and `com.solarmetric.kodo.ResultListProperties` can now be used to specify a custom implementation of the CustomResultList interface that will be used to hold queries.

Notable changes

- The distributed data cache framework has been changed to use the transaction event listener framework to communicate commit information to remote JVMs. This means that deployments that use distributed caching must set up the `com.solarmetric.kodo.RemoteCommitProviderClass` and `com.solarmetric.kodo.RemoteCommitProviderProperties` configuration properties appropriately. Additionally, communication-related configuration properties in the `com.solarmetric.kodo.DataCacheProperties` must be removed.

For example, to configure Kodo to use JMS for distributed commit notification, your properties would look like so: To configure Kodo to just share commit notifications among PersistenceManagerFactories in the

```
com.solarmetric.kodo.DataCacheClass: com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl
com.solarmetric.kodo.RemoteCommitProviderClass: com.solarmetric.kodo.runtime.event.impl.JMSRem
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
like so: To configure Kodo to just share commit notifications among PersistenceManagerFactories in the
same JVM, your properties would look like so:
```

```
com.solarmetric.kodo.DataCacheClass: com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl
com.solarmetric.kodo.RemoteCommitProviderClass: com.solarmetric.kodo.runtime.event.impl.Single
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
same JVM, your properties would look like so:
```

- The UDPCache distributed data cache plug-in has been removed. People interested in using UDP for cache

invalidation should implement the `com.solarmetric.kodo.runtime.event.RemoteCommitProvider` interface.

- The `DataCache` interface and associated implementations have been overhauled in a number of ways. As a result, it is unlikely that previously-created `DataCache` implementations will work with Kodo 2.5.
 - When `IgnoreCache` is set to `false` and a query is executed after modifications to instances that are in the query's access path, Kodo may automatically flush all modifications in the current transaction to the database, and performs the query against the data store. The behavior depends on numerous settings; see `FlushBeforeQueries` for details. Previous releases of Kodo evaluated these types of queries in-memory, which can incur a considerable performance penalty.
 - Added a validation to ensure that ordering strings explicitly use either `ascending` or `descending` correctly, and do not specify any other values for the ordering.
 - Eclipse/WSAD plugin has changed to 1.0.1. The Kodo view is now located in the Java grouping, as opposed to Debug. The plugin is now compatible with Eclipse 2.1. In addition, the required jars in the `plugin.xml` has been changed to include Jakarta's `lang` jar (included with the distribution). The plugin should be reinstalled (remove the old `com.solarmetric...` directory and reinstall according to the documentation).
 - NetBeans/SunONE plugin users should install the Jakarta `lang` jar from the distribution into the `lib/ext` directory of their installation. In addition, `serp.jar` should be removed as it is now part of the regular distribution and is no longer needed. See the full list of required jars in the SunONE/NetBeans portion of the documentation.
- Bugfixes
 - Fixed `datacache` issue with over-eager loading of relations.
 - Fix for potential inefficiency when many threads concurrently access the data cache.
 - Fixed finalization bug in connection pooling that allowed closed connections to be returned from the connection pool.
 - Placed subselects in generated SQL for `isEmpty` on right side of expression to placate DB2.
 - Using persistence-capable parameters that implement `Collection` or `Map` in a JDOQL query now works.
 - Assorted minor bugfixes and error message improvements.
 - Method name misspelling in `com.solarmetric.kodo.impl.jdbc.SQLExecutionListener` has been fixed. As a result, implementations of this interface must be changed to use the correctly-spelled method name.
 - Merged 2.4.3 bugfixes. See below.
 - Fixed many query bugs.

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 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 `PreparedStatement`s and batch updates whenever possible for very significant performance benefits. See the documentation for the `com.solarmetric.kodo.impl.jdbc.UsePreparedStatement`s 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 `GenericDicitionary.toSQL()` and `GenericDicitionary.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()`

ready in time for this release, but are expected to be added to future releases:

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.

Sun JDK 1.2.2 on Solaris

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.

MySQL

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.

PostgreSQL

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`

IBM DB2

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.

SQLServer

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.

Sybase

PointBase

- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending `PointbaseDictionary`

PointBase

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)
Informix Dynamic Server	9.30.UC10	Informix JDBC driver	2.21.JC2
DB2	8.1	IBM DB2 JDBC Universal Driver	1.0.581
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)

Following is a table of the database versions and JDBC driver versions that are supported by Kodo JDO.

G.1. Example properties for Pointbase

```
javax.jdo.option.ConnectionDriverName: \
    com.pointbase.jdbc.jdbcUniversalDriver
javax.jdo.option.ConnectionURL: \
    jdbc:pointbase:DB_NAME, database.home=pointbasedb, create=true, cache.size=10000, database.pagesize=
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.2. Example properties for IBM DB2

```
javax.jdo.option.ConnectionDriverName: com.ibm.db2.jcc.DB2Driver
javax.jdo.option.ConnectionURL: jdbc:db2://SERVER_NAME:SERVER_PORT/DB_NAME
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.3. Example properties for Informix Dynamic Server

```
javax.jdo.option.ConnectionDriverName: com.informix.jdbc.IfxDriver
javax.jdo.option.ConnectionURL: \
    jdbc:informix-sqli://SERVER_NAME:SERVER_PORT/DB_NAME:INFORMIXSERVER=SERVER_ID
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
```

```
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.4. Example properties for Oracle

```
javax.jdo.option.ConnectionDriverName: oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionURL: jdbc:oracle:thin:@SERVER_NAME:1521:DB_NAME
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.5. Example properties for PostgreSQL

```
javax.jdo.option.ConnectionDriverName: org.postgresql.Driver
javax.jdo.option.ConnectionURL: jdbc:postgresql://SERVER_NAME:5432/DB_NAME
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.6. Example properties for SQLServer

```
javax.jdo.option.ConnectionDriverName: \
com.microsoft.jdbc.sqlserver.SQLServerDriver
javax.jdo.option.ConnectionURL: \
jdbc:microsoft:sqlserver://SERVER_NAME:1433;DatabaseName=DB_NAME;SelectMethod=cursor
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.7. Example properties for Sybase

```
javax.jdo.option.ConnectionDriverName: com.sybase.jdbc.SybDriver
javax.jdo.option.ConnectionURL: \
jdbc:sybase:Tds:SERVER_NAME:4100/DB_NAME?ServiceName=DB_NAME&BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.8. Example properties for Hypersonic

```
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionURL: jdbc:hsqldb:DB_NAME
javax.jdo.option.ConnectionUserName: sa
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

G.9. Example properties for MySQL

```
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL: jdbc:mysql://SERVER_NAME/DB_NAME
javax.jdo.option.ConnectionUserName: USERNAME
javax.jdo.option.ConnectionPassword: PASSWORD
com.solarmetric.kodo.LicenseKey: LICENSE_KEY
```

Appendix H. Common Database Errors

Following is a list of known SQL errors, and potential solutions to the problems that they represent.

Table H.1. Known Database Error Codes

Database	Error Code	SQL State	Message	Solution
DB2	-803	23505	SQL0803N One or more values in the INSERT statement, UPDATE statement, or foreign key update caused by a DELETE statement are not valid because the primary key, unique constraint or unique index identified by "1" constrains table "%s" from having duplicate rows for those columns.	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
DB2	-511	42829	SQL0511N The FOR UPDATE clause is not allowed because the table specified by the cursor cannot be modified.	A datastore transaction read was attempted on a table that is marked as read-only. Either read the data outside of a transaction, or use optimistic transactions.
DB2	-401	42818	SQL0401N The data types of the operands for the operation ">=" are not compatible.	A mathematical comparison query was attempted on a field whose mapping was to a non-numeric field, such as VARCHAR. DB2 disallows such queries.
DB2	-302	22003	SQL0302N The value of a host variable in the EXECUTE or OPEN statement is too large for its corresponding use.	Possible attempt to store a String of a length greater than is allowed by the database's column definition. If creation is done via the schematool, ensure

Database	Error Code	SQL State	Message	Solution
				that the column-length metadata field is of sufficient length.
DB2	-204	42S02	SQL0204N "%s" is an undefined name.	The database schema does not match the mapping defined in the metadata for the persistent class. See the metadata documentation.
DB2	-99999	22003	CLI0111E Numeric value out of range.	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
DB2	-99999	HY003	CLI0122E Program type out of range.	A numeric or String range error occurred. Ensure that the capacity of the numeric or string column is sufficient to store the specified value the persistent object is attempting to store.
HSQL	-8	23000	Integrity constraint violation in statement %s	Attempted modification of a row that would cause a violation of referential integrity constraints. Note that cascading deletes can be obtained by using the dependency metadata extensions. Otherwise, it is the responsibility of the application to deal with maintenance of database-specific referential integrity constraints.
HSQL	-9	23000	Violation of unique index: 23000 Violation of unique index in statement %s	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
HSQL	-40	S1000	General error:	A numeric range error

Database	Error Code	SQL State	Message	Solution
			S1000 General error java.lang.NumberFormatException: %d in statement %s	occured. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that some versions of HSQL have a bug that prevents Long.MIN_VALUE from being stored.
MySQL	1062	S1009	Invalid argument value: Duplicate entry '1' for key 1	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
MySQL	1196	S1000	General error: Warning: Some non-transactional changed tables couldn't be rolled back	One or more tables that are being manipulated are not configured to be transactional. Tables in MySQL, by default, do not support transactions. Table type for schema creation can be configured with the TableType parameter of the DictionaryProperties property.
MySQL	1213	S1000	General error: Deadlock found when trying to get lock; Try restarting transaction	A deadlock occurred during a datastore transaction. This can occur when transaction TRANS1 locks table TABLE1, transaction TRANS2 locks table TABLE2, TRANS1 lines up to get a lock on TABLE2, and then TRANS2 lines up to get a lock on TABLE1. Deadlock prevention is the responsibility of the application, or the application server in which it runs. For more details, see the

Database	Error Code	SQL State	Message	Solution
				MySQL deadlock documentation.
MySQL	0	08S01	Communication link failure: java.io.IOException	The TCP connection underlying the JDBC Connection has been closed, possibly due to a timeout. If using Kodo's default DataSource wrapper, connection testing can be configured via the ConnectionTestTimeout property.
MySQL	1030	S1000	General error: Got error 139 from table handler	This is a bug in MySQL server, and can occur when using tables of type InnoDB when long SQL statements are sent to the server. Upgrade to a more recent version of MySQL to resolve the problem.
MySQL	1054	S0022	Column not found: Unknown column 'Infinity' in 'field list'	MySQL disallows storage of Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY values.
Oracle	17069	null	Use explicit XA call	Manual transaction operations were attempted on a DataSource that was configured to use an XA transaction. In order to utilize XA transactions, set the TransactionMode property to xa.
Oracle	17433	null	invalid arguments in call	The Oracle JDBC driver throws this exception when a null username or password were specified. A username and password was not specified in the kodo.properties, nor was it specified in the database configuration mechanism, nor was it specified in the PersistenceManag

Database	Error Code	SQL State	Message	Solution
				<code>er.getPersistenceManager</code> invocation.
Oracle	904	42000	ORA-00904: invalid column name	The database schema does not match the mapping defined in the metadata for the persistent class. See the metadata documentation.
Oracle	1722	42000	ORA-01722: invalid number	A number that Oracle cannot store has been persisted. This can happen when a String field in the persistent class is mapped to an Oracle column of type NUMBER and the String value is not numeric.
Oracle	1000	72000	ORA-01000: maximum open cursors exceeded	<p>Oracle limits the number of statements that can be open at any given time, and the application has made requests that keep open more statements than Oracle can handle. This can be resolved in one of the following ways:</p> <ol style="list-style-type: none"> 1. Increase the number of cursors allowed in the database. This is typically done by increasing the <code>open_cursors</code> parameter in the <code>initSIDNAME.ora</code> file. 2. Ensure that Kodo Query results and Extents are being closed, since open results will maintain an open <code>ResultSet</code> on the server side until they are garbage

Database	Error Code	SQL State	Message	Solution
				collected. 3. If using Kodo's default DataSource implementation, decrease the value of the StatementCache MaxSize parameter.
Oracle	932	42000	ORA-00932: inconsistent datatypes: expected - got CLOB	A normal String field was mapped to an Oracle CLOB type. Oracle requires special handling for CLOBs. Ensure that the metadata for the persistent field is specified as a CLOB by setting the column-length field to -1.
Oracle	1	23000	ORA-00001: unique constraint (%s) violated	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
Oracle	0	null	Underflow Exception	A numeric underflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that Oracle NUMERIC fields have a limitation of 38 digits.
Oracle	0	null	Overflow Exception	A numeric overflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that Oracle NUMERIC fields have

Database	Error Code	SQL State	Message	Solution
				a limitation of 38 digits.
Pointbase	78003	ZW003	The value "%s" cannot be converted to a number.	This can happen when a String field in the persistent class is mapped to a numeric column, and the String value cannot be parsed into a number.
Pointbase	25203	22003	Data exception - numeric value out of range. %d.	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
PostgreSQL	0	null	ERROR: Unable to identify an operator '>=' for types 'numeric' and 'double precision' You will have to retype this query using an explicit cast	An integer field is mapped to a decimal column type. PostgreSQL disallows performing numeric comparisons between integers and decimals.
PostgreSQL	0	null	ERROR: Cannot insert a duplicate key into unique index bug488pcx_pkey	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
PostgreSQL	0	null	ERROR: Attribute 'infinity' not found	PostgreSQL disallows storage of Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY values.
PostgreSQL	0	null	You will have to retype this query using an explicit cast	A String field is mapped to a numeric column type. PostgreSQL disallows performing String comparisons in queries against a numeric column.
SQLServer	0	08007	Can't start a cloned	Append ";SelectMethod=curs

Database	Error Code	SQL State	Message	Solution
			connection while in manual transaction mode.	or" to the ConnectionURL. See the description of the problem on the Microsoft support site.
SQLServer			sp_cursorclose: The cursor identifier value provided (abcdef0) is not valid.	This can sometimes show up as a warning when Kodo is closing a PreparedStatement. It is due to a bug in the SQLServer driver, and can be ignored, since it should not affect anything.
SQLServer	306	HY000	The text, ntext, and image data types cannot be compared or sorted, except when using IS NULL or LIKE operator.	A Query ordering was attempted on a field that is mapped to a CLOB or BLOB, which is disallowed by SQLServer.
SQLServer	8114	HY000	Error converting data type varchar to %s.	This can happen when a String field in the persistent class is mapped to a numeric column, and the String value cannot be parsed into a number.
SQLServer	245	22018	Syntax error converting the varchar value '%s' to a column of data type int.	This can happen when a String field in the persistent class is mapped to a numeric column, and the String value cannot be parsed into a number.
SQLServer	2627	23000	Violation of PRIMARY KEY constraint 'PK_%%s'. Cannot insert duplicate key in object '%%s'.	Duplicate values have been inserted into a primary key column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate primary keys when using application identity.
SQLServer	169	HY000	A column has been specified more than once in the order by	Ensure that there are no duplicates in the ordering of the Query.

Database	Error Code	SQL State	Message	Solution
			list. Columns in the order by list must be unique.	
SQLServer	0	HY000	Object has been closed.	The TCP connection underlying the JDBC Connection may have been closed, possibly due to a timeout. If using Kodo's default DataSource wrapper, connection testing can be configured via the ConnectionTestTimeout property.
Sybase	311	zzzzz	The optimizer could not find a unique index which it could use to scan table '%s' for cursor 'jconnect_implicit_%d'	A pessimistic lock was attempted on a table that does not have a primary key (or other unique index). By default, the Kodo schematool does not create primary keys for join tables. In order to use datastore locking for relations, an IDENTITY column should be added to any tables that do not already have them.
Sybase	2762		The 'CREATE TABLE' command is not allowed within a multi-statement transaction in the 'tempdb' database.	This may happen when running the schematool against a Sybase database that is not configured to allow schema-altering commands to be executed from within a transaction. This can be enabled by entering the command sp_dboption database_name,"ddl in tran",true from isql . See the Sybase documentation for Allowing data definition commands in transactions.
Sybase	0	JZ0BE	JZ0BE: BatchUpdateException: Error	A numeric overflow occurred. Ensure that the capacity of the

Database	Error Code	SQL State	Message	Solution
			occurred while executing batch statement: Arithmetic overflow during implicit conversion of NUMERIC value '%d' to a NUMERIC field.	numeric column is sufficient to store the specified value the persistent object is attempting to store.
Sybase	0	JZ00B	JZ00B: Numeric overflow.	A numeric overflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
Sybase	257	42000	Implicit conversion from datatype 'VARCHAR' to 'TINYINT' is not allowed. Use the CONVERT function to run this query.	A String field is stored in a column of numeric type. Sybase disallows querying against these fields.
Sybase	169	ZZZZZ	Expression '1' and '8' in the ORDER BY list are same. Expressions in the ORDER BY list must be unique.	Ensure that there are no duplicates in the ordering of the Query.
Sybase	511	ZZZZZ	Attempt to update or insert row failed because resultant row of size 2009 bytes is larger than the maximum size (1961 bytes) allowed for this table.	Attempt to store a String of a length greater than is allowed by the database's column definition. If creation is done via the schematool, ensure that the column-length metadata field is of sufficient length.
Sybase	2601	23000	Attempt to insert duplicate key row in object '%s' with unique index '%s'	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.