

## **SolarMetric Kodo™ JDO 3.3.5 Developers Guide**

**Copyright © 2001 - 2005 SolarMetric Inc.**

---

# **SolarMetric Kodo™ JDO 3.3.5 Developers Guide**

Copyright © 2001 - 2005 SolarMetric Inc.

---

---

---

# Table of Contents

I. Introduction .....	1
1. SolarMetric Kodo JDO .....	3
1.1. About This Document .....	3
2. SolarMetric Kodo JDO Installation .....	4
2.1. Overview .....	4
2.2. Updates .....	4
2.3. Key Files in the download .....	4
2.4. Quick Start .....	4
2.5. Upgrading from Kodo 2 .....	5
2.6. Requirements .....	5
2.7. Terminology .....	5
2.8. Windows Installation (installer) .....	5
2.9. Windows Installation (no installer -- zip file) .....	5
2.10. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation .....	6
2.11. Borland JBuilder .....	6
2.12. Installing Kodo into JBuilder .....	6
2.13. Common installation problems .....	6
2.14. Resources .....	7
2.15. Sales Inquiries .....	7
II. Java Data Objects .....	8
1. Introduction .....	11
1.1. Intended Audience .....	11
1.2. Transparent Persistence .....	11
2. Why JDO? .....	12
3. JDO Architecture .....	14
3.1. JDO Exceptions .....	15
4. PersistenceCapable .....	16
4.1. JDO Enhancer .....	16
4.2. Persistence-Capable vs. Persistence-Aware .....	17
4.3. Restrictions on Persistent Classes .....	17
4.3.1. Default or No-Arg Constructor .....	17
4.3.2. Inheritance .....	17
4.3.3. Persistent Fields .....	18
4.3.4. Conclusions .....	20
4.4. InstanceCallbacks .....	20
4.5. JDO Identity .....	21
4.5.1. Datastore Identity .....	22
4.5.2. Application Identity .....	23
4.5.2.1. Application Identity Hierarchies .....	25
4.5.3. Single Field Identity .....	26
4.6. Conclusions .....	26
5. Metadata .....	27
5.1. Metadata DTD .....	27
5.2. Metadata Placement .....	31
6. JDOHelper .....	33
6.1. Persistence-Capable Operations .....	33
6.2. Lifecycle Operations .....	33
6.3. PersistenceManagerFactory Construction .....	36
7. PersistenceManagerFactory .....	37
7.1. Obtaining a PersistenceManagerFactory .....	37
7.2. PersistenceManagerFactory Properties .....	37
7.2.1. Connection Configuration .....	38
7.2.2. PersistenceManager and Transaction Defaults .....	39

---

7.3. Obtaining PersistenceManagers .....	40
7.4. Properties and Supported Options .....	41
8. PersistenceManager .....	43
8.1. User Object Association .....	44
8.2. Configuration Properties .....	44
8.3. Transaction Association .....	44
8.4. Persistence-Capable Lifecycle Management .....	44
8.5. Lifecycle Examples .....	46
8.6. JDO Identity Management .....	47
8.7. Extent Factory .....	48
8.8. Query Factory .....	48
8.9. Closing .....	49
9. Transaction .....	50
9.1. Transaction Types .....	50
9.2. The JDO Transaction Interface .....	51
10. Extent .....	53
11. Query .....	54
11.1. Object Filtering .....	54
11.2. JDOQL .....	56
11.3. Advanced Object Filtering .....	58
11.4. Compiling and Executing Queries .....	61
11.5. Limits and Ordering .....	62
11.6. Projections .....	64
11.7. Aggregates .....	66
11.8. Result Class .....	68
11.8.1. JavaBean Result Class .....	69
11.8.2. Generic Result Class .....	70
11.9. Single-String JDOQL .....	70
11.10. Named Queries .....	72
11.10.1. Named Query DTD .....	72
11.10.2. Named Query Examples .....	73
11.11. Conclusion .....	75
12. SQL Queries .....	76
12.1. Creating SQL Queries .....	76
12.2. Retrieving Persistent Objects with SQL .....	77
12.3. SQL Projections .....	78
12.4. Named SQL Queries .....	79
12.5. Conclusion .....	80
13. Conclusion .....	81
III. Kodo JDO Tutorials .....	82
1. Kodo JDO Tutorials .....	84
1.1. Tutorial Requirements .....	84
2. Kodo JDO Tutorial .....	85
2.1. The Pet Shop .....	85
2.1.1. Included Files .....	85
2.1.2. Important Utilities .....	86
2.2. Getting Started .....	86
2.2.1. Configuring the Data Store .....	87
2.3. Inventory Maintenance .....	88
2.3.1. Persisting Objects .....	89
2.3.2. Deleting Objects .....	90
2.4. Inventory Growth .....	91
2.5. Behavioral Analysis .....	92
2.5.1. Complex Queries .....	96
2.6. Extra Features .....	97
3. Reverse Mapping Tool Tutorial .....	98
3.1. Magazine Shop .....	98
3.2. Setup .....	98
3.2.1. Tutorial Files .....	98

---

3.2.2. Important Utilities .....	99
3.3. Generating Persistent Classes .....	99
3.4. Using the Finder .....	101
4. J2EE Tutorial .....	103
4.1. Prerequisites for the Kodo J2EE Tutorial .....	103
4.2. J2EE Installation Types .....	103
4.3. Installing Kodo JCA .....	103
4.3.1. JBoss 3.0 .....	103
4.3.2. JBoss 3.2 .....	104
4.3.3. WebLogic 6.1 to 7.x .....	104
4.3.4. WebLogic 8.1 .....	105
4.3.5. WebSphere 5 .....	105
4.3.6. SunONE Application Server 7 / Sun Java Enterprise Server 8 - 8.1 .....	106
4.3.7. Macromedia JRun 4 .....	106
4.3.8. Borland Enterprise Server 5.2 - 6.0 .....	107
4.3.9. Non-JCA Application Server Deployment .....	107
4.4. Installing the J2EE Sample Application .....	108
4.4.1. Compiling and Building The Sample Application .....	109
4.4.2. Deploying Sample To JBoss .....	109
4.4.3. Deploying Sample To WebLogic 6.1 to 7.x .....	110
4.4.4. Deploying Sample To WebLogic 8.1 .....	110
4.4.5. Deploying Sample To SunONE / JES .....	110
4.4.6. Deploying Sample To JRun .....	110
4.4.7. Deploying Sample To WebSphere .....	110
4.4.8. Deploying Sample To Borland Enterprise Server 5.2 .....	111
4.5. Using The Sample Application .....	111
4.6. Sample Architecture .....	111
4.7. Code Notes and J2EE Tips .....	112
IV. Kodo JDO Frequently Asked Questions .....	114
1. Kodo JDO Frequently Asked Questions .....	116
1.1. General .....	116
1.2. Database .....	117
1.3. Programming with Kodo .....	119
1.4. How do I ... ? .....	122
1.5. Common errors .....	124
1.6. Productivity tools .....	124
1.7. Performance .....	125
1.8. Scalability .....	126
1.9. Application servers .....	126
1.10. Locking .....	127
1.11. Transactions .....	128
V. Kodo JDO Reference Guide .....	129
1. Introduction .....	138
1.1. Intended Audience .....	138
2. Configuration .....	139
2.1. Introduction .....	139
2.2. Runtime Configuration .....	139
2.3. Command Line Configuration .....	139
2.3.1. Code Formatting .....	140
2.4. Plugin Configuration .....	140
2.5. JDO Standard Properties .....	141
2.5.1. javax.jdo.PersistenceManagerFactoryClass .....	141
2.5.2. javax.jdo.option.ConnectionDriverName .....	142
2.5.3. javax.jdo.option.ConnectionFactoryName .....	142
2.5.4. javax.jdo.option.ConnectionFactory2Name .....	142
2.5.5. javax.jdo.option.ConnectionPassword .....	142
2.5.6. javax.jdo.option.ConnectionURL .....	143
2.5.7. javax.jdo.option.ConnectionUserName .....	143
2.5.8. javax.jdo.option.IgnoreCache .....	143

2.5.9. javax.jdo.option.Multithreaded .....	143
2.5.10. javax.jdo.option.NontransactionalRead .....	143
2.5.11. javax.jdo.option.NontransactionalWrite .....	144
2.5.12. javax.jdo.option.Optimistic .....	144
2.5.13. javax.jdo.option.RestoreValues .....	144
2.5.14. javax.jdo.option.RetainValues .....	144
2.6. Kodo JDO Properties .....	145
2.6.1. kodo.AggregateListeners .....	145
2.6.2. kodo.ClassResolver .....	145
2.6.3. kodo.ConnectionProperties .....	145
2.6.4. kodo.ConnectionFactoryProperties .....	145
2.6.5. kodo.Connection2DriverName .....	146
2.6.6. kodo.Connection2Password .....	146
2.6.7. kodo.Connection2URL .....	146
2.6.8. kodo.Connection2UserName .....	146
2.6.9. kodo.Connection2Properties .....	147
2.6.10. kodo.ConnectionFactory2Properties .....	147
2.6.11. kodo.ConnectionRetainMode .....	147
2.6.12. kodo.CopyObjectIds .....	147
2.6.13. kodo.DataCache .....	148
2.6.14. kodo.DataCacheTimeout .....	148
2.6.15. kodo.DynamicDataStructs .....	148
2.6.16. kodo.EagerFetchMode .....	148
2.6.17. kodo.FetchBatchSize .....	149
2.6.18. kodo.FetchGroups .....	149
2.6.19. kodo.FilterListeners .....	149
2.6.20. kodo.FlushBeforeQueries .....	149
2.6.21. kodo.InverseManager .....	149
2.6.22. kodo.LicenseKey .....	150
2.6.23. kodo.LockManager .....	150
2.6.24. kodo.LockTimeout .....	150
2.6.25. kodo.Log .....	150
2.6.26. kodo.ManagedRuntime .....	151
2.6.27. kodo.ManagementConfiguration .....	151
2.6.28. kodo.MetadataLoader .....	151
2.6.29. kodo.ObjectLookupMode .....	151
2.6.30. kodo.OrphanedKeyAction .....	152
2.6.31. kodo.PersistenceManagerImpl .....	152
2.6.32. kodo.PersistenceManagerServer .....	152
2.6.33. kodo.PersistentClasses .....	152
2.6.34. kodo.ProxyManager .....	153
2.6.35. kodo.QueryCache .....	153
2.6.36. kodo.QueryCompilationCache .....	153
2.6.37. kodo.ReadLockLevel .....	153
2.6.38. kodo.RemoteCommitProvider .....	154
2.6.39. kodo.RestoreMutableValues .....	154
2.6.40. kodo.RetainValuesInOptimistic .....	154
2.6.41. kodo.RetryClassRegistration .....	154
2.6.42. kodo.SubclassFetchMode .....	155
2.6.43. kodo.TransactionMode .....	155
2.6.44. kodo.WriteLockLevel .....	155
2.6.45. kodo.jdbc.ClassIndicator .....	155
2.6.46. kodo.jdbc.ConnectionDecorators .....	156
2.6.47. kodo.jdbc.DataSourceMode .....	156
2.6.48. kodo.jdbc.DBDictionary .....	156
2.6.49. kodo.jdbc.FetchDirection .....	156
2.6.50. kodo.jdbc.ForeignKeyConstraints .....	157
2.6.51. kodo.jdbc.JDBCListeners .....	157
2.6.52. kodo.jdbc.LRSSize .....	157

2.6.53. kodo.jdbc.MappingFactory .....	157
2.6.54. kodo.jdbc.ResultSetType .....	158
2.6.55. kodo.jdbc.SchemaFactory .....	158
2.6.56. kodo.jdbc.Schemas .....	158
2.6.57. kodo.jdbc.SequenceFactory .....	158
2.6.58. kodo.jdbc.SubclassMapping .....	159
2.6.59. kodo.jdbc.SynchronizeMappings .....	159
2.6.60. kodo.jdbc.TransactionIsolation .....	159
2.6.61. kodo.jdbc.UpdateManager .....	159
2.6.62. kodo.jdbc.VersionIndicator .....	159
3. Logging .....	161
3.1. Logging Channels .....	161
3.2. Kodo Logging .....	162
3.3. Disabling Logging .....	163
3.4. Log4J .....	163
3.5. Apache Commons Logging .....	164
3.5.1. JDK 1.4 java.util.logging .....	164
3.6. Custom Log .....	165
4. JDBC .....	166
4.1. Using the Kodo JDO DataSource .....	166
4.2. Using a Third-Party DataSource .....	168
4.2.1. Enlisted Data Sources .....	169
4.3. Database Support .....	169
4.3.1. MySQLDictionary parameters .....	175
4.3.2. OracleDictionary parameters .....	175
4.4. Configuring the DBDictionary .....	176
4.5. Accessing Multiple Databases .....	176
4.6. Setting the Transaction Isolation .....	176
4.7. Setting the SQL Join Syntax .....	177
4.8. Configuring the Use of JDBC Connections .....	177
4.9. Runtime Access to JDBC Connections .....	179
4.10. Large Result Sets .....	180
4.11. SQL Statement Ordering & Foreign Keys .....	182
5. Persistent Classes .....	184
5.1. Restrictions on Persistent Classes .....	184
5.2. Object Identity .....	184
5.2.1. Datastore Identity .....	184
5.2.2. Application Identity .....	184
5.2.3. Single Field Identity .....	185
5.2.4. Primary Key Generation .....	185
5.2.4.1. Sequence Factory .....	185
5.2.4.2. Auto-Increment .....	187
5.2.4.3. Sequence-Assigned .....	188
5.3. Managed Inverses .....	188
5.4. Mutable Second Class Object Fields .....	189
5.4.1. Restoring Mutable Fields .....	189
5.4.2. Typing and Ordering .....	189
5.4.3. Proxies .....	190
5.4.3.1. Smart Proxies .....	190
5.4.3.2. Large Result Set Proxies .....	190
5.4.3.3. Custom Proxies .....	191
5.5. Enhancement .....	192
5.6. Auto-Generating Classes from a Schema .....	193
5.6.1. Customizing Reverse Mapping .....	195
5.7. Persistent Class List .....	196
6. Metadata .....	197
6.1. Generating Default JDO Metadata .....	197
6.2. JDO Metadata Extensions .....	197
6.2.1. Relation Extensions .....	198

6.2.1.1. inverse-owner .....	198
6.2.1.2. inverse-logical .....	198
6.2.1.3. dependent .....	198
6.2.1.4. element-dependent .....	199
6.2.1.5. value-dependent .....	199
6.2.1.6. key-dependent .....	199
6.2.1.7. type .....	199
6.2.1.8. element-type .....	199
6.2.1.9. value-type .....	200
6.2.1.10. key-type .....	200
6.2.1.11. lrs .....	200
6.2.1.12. Example .....	200
6.2.2. Schema Extensions .....	200
6.2.2.1. jdbc-size .....	200
6.2.2.2. jdbc-element-size .....	200
6.2.2.3. jdbc-value-size .....	201
6.2.2.4. jdbc-key-size .....	201
6.2.2.5. jdbc-type .....	201
6.2.2.6. jdbc-sql-type .....	201
6.2.2.7. jdbc-indexed .....	201
6.2.2.8. jdbc-element-indexed .....	201
6.2.2.9. jdbc-value-indexed .....	201
6.2.2.10. jdbc-key-indexed .....	201
6.2.2.11. jdbc-ref-indexed .....	201
6.2.2.12. jdbc-version-ind-indexed .....	202
6.2.2.13. jdbc-class-ind-indexed .....	202
6.2.2.14. jdbc-delete-action .....	202
6.2.2.15. jdbc-element-delete-action .....	202
6.2.2.16. jdbc-value-delete-action .....	203
6.2.2.17. jdbc-key-delete-action .....	203
6.2.2.18. jdbc-ref-delete-action .....	203
6.2.2.19. Example .....	203
6.2.3. Object-Relational Mapping Extensions .....	203
6.2.3.1. jdbc-class-map-name .....	203
6.2.3.2. jdbc-version-ind-name .....	204
6.2.3.3. jdbc-class-ind-name .....	204
6.2.3.4. jdbc-field-map-name .....	204
6.2.3.5. jdbc-field-mappings .....	204
6.2.3.6. jdbc-ordered .....	204
6.2.3.7. jdbc-container-meta .....	204
6.2.3.8. jdbc-null-ind .....	205
6.2.3.9. externalizer .....	205
6.2.3.10. factory .....	205
6.2.3.11. external-values .....	205
6.2.3.12. jdbc-class-ind-value .....	205
6.2.3.13. Example .....	205
6.2.4. Miscellaneous Extensions .....	206
6.2.4.1. detachable .....	206
6.2.4.2. detached-objectid-field .....	206
6.2.4.3. detached-state-field .....	206
6.2.4.4. fetch-group .....	206
6.2.4.5. lock-group .....	206
6.2.4.6. lock-groups .....	206
6.2.4.7. data-cache .....	206
6.2.4.8. data-cache-timeout .....	206
6.2.4.9. sequence-assigned .....	206
6.2.4.10. subclass-fetch-mode .....	207
6.2.4.11. eager-fetch-mode .....	207
6.2.4.12. jdbc-sequence-factory .....	207

6.2.4.13. jdbc-sequence-name .....	207
6.2.4.14. jdbc-auto-increment .....	207
6.2.4.15. Example .....	207
7. Object-Relational Mapping .....	209
7.1. Mapping Tool .....	209
7.1.1. Using the Mapping Tool .....	210
7.1.2. Generating DDL SQL .....	212
7.2. Automatic Runtime Mapping .....	213
7.3. Mapping Factory .....	214
7.3.1. Importing and Exporting Mapping Data .....	215
7.4. Mapping File XML Format .....	215
7.5. Mapping Notes .....	217
7.5.1. Join Attributes .....	218
7.5.2. Non-Standard Joins .....	218
7.6. Class Mapping .....	219
7.6.1. Base Mapping .....	220
7.6.2. Flat Inheritance Mapping .....	220
7.6.2.1. Advantages of using Flat Inheritance Mapping .....	222
7.6.2.2. Disadvantages of using Flat Inheritance Mapping .....	222
7.6.3. Vertical Inheritance Mapping .....	222
7.6.3.1. Advantages of using Vertical Inheritance Mapping .....	223
7.6.3.2. Disadvantages of using Vertical Inheritance Mapping .....	224
7.6.3.3. Vertical Select Modes .....	224
7.6.4. Horizontal Inheritance Mapping .....	224
7.6.4.1. Special considerations when using Horizontal Inheritance Mapping .....	228
7.6.4.2. Advantages of using Horizontal Inheritance Mapping .....	228
7.6.4.3. Disadvantages of using Horizontal Inheritance Mapping .....	228
7.6.5. Custom Class Mapping .....	228
7.7. Version Indicator .....	229
7.7.1. Version Number Indicator .....	229
7.7.2. Version Date Indicator .....	230
7.7.3. State Image Indicator .....	231
7.7.4. Custom Version Indicator .....	232
7.8. Class Indicator .....	232
7.8.1. In-Class-Name Indicator .....	232
7.8.2. Metadata Value Indicator .....	233
7.8.3. Subclass-Join Indicator .....	235
7.8.4. Custom Class Indicator .....	236
7.9. Field Mapping .....	236
7.9.1. Value Mapping .....	236
7.9.2. Blob Mapping .....	238
7.9.3. Clob Mapping .....	240
7.9.4. Byte Array Mapping .....	241
7.9.5. One-to-One Mapping .....	242
7.9.6. PC One-to-One Mapping .....	246
7.9.7. Embedded One-to-One Mapping .....	248
7.9.8. Enumeration Mapping .....	250
7.9.9. Collection Mapping .....	251
7.9.10. Many-to-Many Mapping .....	253
7.9.11. One-to-Many Mapping .....	256
7.9.12. PC Collection Mapping .....	259
7.9.13. Map Mapping .....	260
7.9.14. N-to-Many Map Mapping .....	261
7.9.15. Many-to-N Map Mapping .....	263
7.9.16. Many-to-Many Map Mapping .....	264
7.9.17. PC Map Mapping .....	266
7.9.18. N-to-PC Map Mapping .....	267
7.9.19. PC-to-N Map Mapping .....	269
7.9.20. PC-to-Many Map Mapping .....	270

7.9.21. Many-to-PC Map Mapping .....	272
7.9.22. Custom Field Mapping .....	273
7.9.23. Externalization .....	273
7.9.24. External Values .....	276
8. Schema Information .....	278
8.1. Schema Reflection .....	278
8.1.1. Schemas List .....	278
8.1.2. Schema Factory .....	278
8.1.3. Schema Generator .....	279
8.2. Schema Tool .....	280
8.3. XML Schema Format .....	282
8.4. The SQLLine Utility .....	284
9. Runtime Deployment .....	286
9.1. JDOHelper .....	286
9.2. KodoHelper .....	286
9.3. J2EE Deployment .....	287
10. JDO Runtime Extensions .....	288
10.1. KodoPersistenceManagerFactory .....	288
10.2. KodoPersistenceManager .....	288
10.2.1. JDO Transaction Events .....	288
10.2.2. JDO 2 Preview Methods .....	288
10.2.3. Lifecycle Events .....	289
10.2.4. PersistenceManager Extension .....	289
10.3. KodoExtent .....	290
10.4. KodoQuery .....	290
10.5. Fetch Configuration .....	290
10.6. KodoHelper .....	290
10.7. Query Extensions .....	290
10.7.1. JDOQL Extensions .....	291
10.7.1.1. Included JDOQL Extensions .....	291
10.7.1.2. Developing Custom JDOQL Extensions .....	292
10.7.1.3. Configuring JDOQL Extensions .....	292
10.7.2. Aggregate Extensions .....	292
10.7.2.1. Configuring Query Aggregates .....	292
10.8. Object Locking .....	292
10.8.1. Configuring Default Locking .....	293
10.8.2. Configuring Lock Levels at Runtime .....	293
10.8.3. Object Locking APIs .....	293
10.8.4. Lock Manager .....	294
10.8.5. Rules for Locking Behavior .....	295
10.8.6. Known Issues and Limitations .....	295
10.9. Orphaned Keys .....	296
11. Remote and Offline JDO .....	297
11.1. Detach and Attach .....	297
11.1.1. Declaring Detachability .....	297
11.1.2. Detach and Attach Behavior .....	298
11.1.3. Defining the Detached Object Graph .....	299
11.1.4. Detach and Attach Callbacks .....	301
11.1.5. Automatic Detachment .....	301
11.1.5.1. Detach on Close .....	301
11.1.5.2. Detach on Serialize .....	302
11.2. Remote Persistence Managers .....	303
11.2.1. Standalone Persistence Manager Server .....	304
11.2.2. HTTP Persistence Manager Server .....	305
11.2.3. Client Persistence Managers .....	305
11.2.4. Data Compression and Filtering .....	307
11.2.5. Remote Persistence Manager Deployment .....	308
12. Management and Monitoring .....	309
12.1. Configuration .....	309

12.1.1. Optional Parameters in Management Group .....	311
12.1.2. Optional Parameters in Remote Group .....	311
12.1.3. Optional Parameters in JSR 160 Group .....	312
12.1.4. Optional Parameters in WebLogic 8.1 Group .....	312
12.1.5. Configuring Logging for Management / Monitoring .....	313
12.2. Kodo Management Console .....	313
12.2.1. Remote Connection .....	313
12.2.1.1. Connecting to Kodo under WebLogic 8.1 .....	314
12.2.1.2. Connecting to Kodo under JBoss 3.2 .....	314
12.2.1.3. Connecting to Kodo under JBoss 4 .....	315
12.2.2. Using the Kodo Management Console .....	315
12.2.2.1. JMX Explorer .....	316
12.2.2.1.1. Executing Operations .....	316
12.2.2.1.2. Listening to Notifications .....	317
12.2.2.2. MBean Panel .....	317
12.2.2.2.1. Notifications / Statistics .....	317
12.2.2.2.2. Setting Attributes .....	317
12.3. Accessing the MBeanServer from Code .....	317
12.4. MBeans .....	318
12.4.1. Log MBean .....	318
12.4.2. Kodo Pooling DataSource MBean .....	318
12.4.3. PreparedStatement Cache MBean .....	318
12.4.4. Query Cache MBean .....	318
12.4.5. Datastore Cache MBean .....	318
12.4.6. TimeWatch MBean .....	318
12.4.7. Runtime MBean .....	319
12.4.8. Profiling MBean .....	319
13. Enterprise Edition .....	321
13.1. Integrating with the Transaction Manager .....	321
13.2. XA Transactions .....	321
13.2.1. Requirements for Using Kodo with XA Transactions .....	322
13.2.2. Configuring Kodo to Utilize XA Transactions .....	322
13.3. JDOQL Subqueries .....	323
13.3.1. Subquery Parameters, Variables, and Imports .....	324
13.4. Direct SQL Execution .....	324
13.5. MethodQL .....	324
13.6. Remote PersistenceManagers .....	325
13.7. Custom Class Mappings .....	325
13.8. Non-relational Database Access .....	325
14. Performance Pack .....	326
14.1. SQL Batching .....	326
14.2. Eager Fetching .....	326
14.2.1. Configuring Eager Fetching .....	327
14.2.2. Eager Fetching Considerations .....	328
14.3. Datastore Cache .....	329
14.3.1. Overview of Kodo JDO Datastore Caching .....	329
14.3.2. Kodo JDO Cache Usage .....	329
14.3.3. Query Caching .....	332
14.3.4. DataCache Integrations .....	334
14.3.5. Cache Extension .....	334
14.3.6. Important Notes .....	334
14.3.7. Known Issues and Limitations .....	335
14.4. Remote Event Notification Framework .....	335
14.4.1. Remote Commit Provider Configuration .....	336
14.4.2. Customization .....	337
14.5. Fetch Groups .....	337
14.5.1. Normal Default Fetch Group Behavior .....	337
14.5.2. Kodo JDO Fetch Group Behavior .....	338
14.5.3. Custom Fetch Group Configuration .....	339

14.5.4. Per-field Fetch Configuration .....	339
14.6. Lock Groups .....	340
14.6.1. Lock Groups and Subclasses .....	341
14.6.2. Lock Group Mapping .....	342
14.7. Profiling .....	342
14.7.1. Profiling in an embedded GUI .....	343
14.7.2. Dumping profiling data to disk from a batch process .....	345
14.7.3. Controlling how the profiler obtains context information .....	345
15. Third Party Integration .....	346
15.1. Overview of Third Party Integration features in Kodo .....	346
15.2. Apache Ant .....	346
15.2.1. Common Ant Configuration Options .....	346
15.2.2. JDO Enhancer Ant Task .....	348
15.2.3. Application Identity Tool Ant Task .....	348
15.2.4. JDO Metadata Tool Ant Task .....	349
15.2.5. Mapping Tool Ant Task .....	349
15.2.6. Reverse Mapping Tool Ant Task .....	349
15.2.7. Schema Tool Ant Task .....	350
15.2.8. Schema Generator Ant Task .....	350
15.3. XDoclet .....	351
15.4. Borland JBuilder .....	355
15.4.1. Installing Kodo Into JBuilder .....	355
15.4.2. Kodo Configuration from JBuilder .....	356
15.4.3. Creating and building JDO projects in JBuilder .....	356
15.4.4. Editing JDO Metadata from JBuilder .....	356
15.4.5. Editing Mapping Info from JBuilder .....	356
15.4.6. JBuilder Project Sample .....	357
15.5. Sun ONE Studio / NetBeans IDE .....	357
15.5.1. Before Installing Kodo into the IDE .....	357
15.5.2. Installing Kodo into the IDE .....	358
15.5.3. Configuring the Kodo Module .....	358
15.5.4. Kodo Template Wizards .....	358
15.5.5. JDO DataObject .....	359
15.5.6. Mapping DataObject .....	359
15.5.7. Kodo Integration into the Build Process .....	359
15.5.8. SunONE / NetBeans Sample .....	359
15.6. Eclipse / WebSphere Studio Integration .....	360
15.6.1. Installing the Kodo Eclipse Plugin .....	360
15.6.2. Configuring the Plugin .....	361
15.6.3. Using Kodo in Eclipse IDEs .....	361
15.6.4. Eclipse Sample .....	361
16. Optimization Techniques .....	362
VI. Kodo JDO Examples .....	367
1. Kodo Sample Code .....	369
1.1. Using Application Identity .....	369
1.2. Using JDO with Java Server Pages (jsp) .....	369
1.3. Custom Proxies .....	369
1.4. JDO Enterprise Java Beans Facade .....	369
1.5. Customizing Logging .....	369
1.6. Custom Sequence Factory .....	369
1.7. Horizontal Mappings .....	369
1.8. Using Externalization to Persist Second Class Objects .....	370
1.9. Using Persistent Classes Without Enhancement .....	370
1.10. XDoclet Integration .....	370
1.11. Custom Mappings .....	370
1.12. Example of full-text searching in JDO .....	371
1.13. JMX Management .....	371
1.14. XML Store Manager .....	372
1.15. Sample Human Resources Model .....	373

1.16. Sample School Schedule Model .....	373
VII. Kodo Development Workbench Guide .....	375
1. Introduction to the Kodo Development Workbench .....	ccclxxviii
1.1. Kodo Development Workbench Requirements .....	ccclxxviii
1. Running and Configuring Kodo Development Workbench .....	379
1.1. Starting Kodo Development Workbench From the Command Line .....	379
1.2. Configuring Kodo Development Workbench .....	379
2. Getting Started with Kodo Development Workbench .....	380
2.1. Beginning The Kodo Development Workbench Tutorial .....	380
2.2. Getting Familiar with Kodo Development Workbench .....	380
2.3. The MetaData Explorer .....	380
2.3.1. MetaData Explorer Basics .....	381
2.3.2. Creating JDO MetaData .....	381
2.3.3. Using The MetaData Explorer .....	381
2.3.4. Editing MetaData .....	382
2.3.5. Additional MetaData Explorer Actions .....	382
2.4. The Schema Explorer .....	382
2.5. Kodo Development Workbench Logging .....	384
2.6. The Details Pane .....	384
2.7. The Editor .....	384
2.7.1. Editor Overview .....	384
2.7.2. Using The Visualization Editor .....	384
2.7.3. Editing Visualization Mappings .....	386
2.7.4. Mapping Fields .....	387
2.7.5. Completing Visualization Changes .....	387
2.7.6. JDOQLEditor .....	387
2.8. Running The Tutorial .....	389
3. Root MetaData Actions .....	390
3.1. Mount JDO File .....	390
3.2. Unmount Files... .....	390
3.3. Create MetaData .....	390
3.4. Import Mapping Info .....	390
4. MetaData Actions .....	391
4.1. Enhance .....	391
4.2. Edit MetaData .....	391
4.3. Add - Recreate Mapping Info .....	391
4.4. Export Mapping Info .....	391
4.5. Drop Mapping Info .....	391
4.6. Remove MetaData .....	391
4.7. Build Schema From Mapping .....	391
4.8. Visualize Mapping .....	391
5. Root Schema Actions .....	394
5.1. Run SchemaTool .....	394
5.2. Refresh Schema From DB .....	394
5.3. Create DB Script .....	394
5.4. Create Change Script .....	394
5.5. Reverse Map Schema .....	395
5.6. Import .schema file .....	395
5.7. Export to .schema file .....	395
6. Schema Actions .....	396
6.1. Drop Schema Object .....	396
6.2. Edit Table .....	396
6.3. Add .....	396
7. The Editors .....	397
7.1. MetaData Editor .....	397
7.2. The Mapping Editor .....	397
7.3. The Visualization Editor .....	398
7.4. The Table Editor .....	398
7.5. The Foreign Keys Editor .....	398

8. JDOQL Editor .....	399
8.1. Query Validator .....	399
8.2. Candidate Class Editor .....	400
8.3. Filter Editor .....	400
8.4. Additional Query Component Editors .....	400
8.4.1. Ordering Editor .....	400
8.4.2. Parameters Editor .....	400
8.4.3. Imports Editor .....	401
8.4.4. Variables Editor .....	401
8.4.5. Aggregates and Projections Editor .....	401
8.4.6. Fetch Configuration Editor .....	401
8.5. Execute Query .....	401
8.6. Show SQL .....	401
8.7. Clear Query .....	401
8.8. Show Java .....	401
8.9. Save Query .....	402
8.10. Load Query .....	402
8.11. Recent Queries .....	402
8.12. Results Browser .....	402
A. JDO Resources .....	403
B. Supported Databases .....	404
B.1. Apache Derby .....	404
B.1.1. Known issues with Derby .....	404
B.2. Borland Interbase .....	405
B.2.1. Known issues with Interbase .....	405
B.3. JDataStore .....	405
B.4. IBM DB2 .....	405
B.4.1. Known issues with DB2 .....	406
B.5. Empress .....	406
B.5.1. Known issues with Empress .....	406
B.6. Hypersonic .....	406
B.6.1. Known issues with Hypersonic .....	406
B.7. Firebird .....	407
B.7.1. Known issues with Firebird .....	407
B.8. Informix .....	407
B.8.1. Known issues with Informix .....	407
B.9. InterSystems Cache .....	407
B.9.1. Known issues with InterSystems Cache .....	408
B.10. Microsoft Access .....	408
B.10.1. Known issues with Microsoft Access .....	408
B.11. Microsoft SQL Server .....	408
B.11.1. Known issues with SQL Server .....	408
B.12. Microsoft FoxPro .....	409
B.12.1. Known issues with Microsoft FoxPro .....	409
B.13. MySQL .....	409
B.13.1. Known issues with MySQL .....	409
B.14. Oracle .....	410
B.14.1. Known issues with Oracle .....	410
B.15. Pointbase .....	410
B.15.1. Known issues with Pointbase .....	410
B.16. PostgreSQL .....	411
B.16.1. Known issues with PostgreSQL .....	411
B.17. Sybase Adaptive Server .....	411
B.17.1. Known issues with Sybase .....	411
C. Common Database Errors .....	413
D. Upgrading Kodo .....	422
D.1. Migrating from Kodo 2 to Kodo 3 .....	422
D.1.1. Source Code Migration .....	422
D.1.1.1. Package Structure Changes .....	422

D.1.1.2. API Changes .....	423
D.1.2. JDO Metadata Migration .....	423
D.1.3. Properties File Migration .....	425
D.1.4. Storing Object-Relational Mapping Data .....	426
D.1.5. Kodo 3 Development Process .....	426
D.2. Migrating from Kodo 3.0 to Kodo 3.1 .....	426
D.3. Migrating from Kodo 3.1 to Kodo 3.2 .....	426
E. Implementation Notes .....	427
E.1. jdoFlags Fields in the Default Fetch Group .....	427
F. DataCache Integrations .....	428
F.1. Tangosol Integration .....	428
F.2. GemStone Gemfire Integration .....	428
G. Development and Runtime Libraries .....	430
H. Release Notes .....	432
I. Known Bugs and Limitations .....	477
Index .....	478

---

# List of Tables

- 2.1. Persistence Mechanisms ..... 12
- 6.1. JDOHelper Lifecycle Methods ..... 35
- 2.1. Pre-defined aliases ..... 153
- 4.1. Validation SQL Defaults ..... 167
- 4.2. Kodo Automatic Flush Behavior ..... 178
- 7.1. Externalizer Options ..... 274
- 7.2. Factory Options ..... 274
- 14.1. Data access methods ..... 329
- 16.1. Optimization Techniques ..... 362
- 4.1. Graph Edges ..... 392
- B.1. Supported Databases and JDBC Drivers ..... 404
- C.1. Known Database Error Codes ..... 413
- D.1. Notable Package Changes ..... 422

---

## List of Examples

3.1. Interaction of JDO Interfaces .....	14
4.1. PersistenceCapable Class .....	16
4.2. Accessing Mutable Persistent Fields .....	19
4.3. Using the InstanceCallbacks Interface .....	21
4.4. JDO Identity Objects .....	22
4.5. Application Identity Class .....	24
5.1. Basic Structure of Metadata Documents .....	27
5.2. Metadata Class Listings .....	29
5.3. Complete Metadata Document .....	31
6.1. Obtaining a PersistenceManagerFactory .....	36
8.1. Persisting Objects .....	46
8.2. Updating Objects .....	46
8.3. Deleting Objects .....	47
9.1. Grouping Operations with Transactions .....	52
10.1. Iterating an Extent .....	53
11.1. Filtering .....	55
11.2. Relation Traversal and Mathematical Operations .....	57
11.3. Precedence, Logical Operators, and String Functions .....	57
11.4. Collections .....	57
11.5. Static Methods .....	58
11.6. Imports and Declared Parameters .....	59
11.7. Implicit Parameters .....	59
11.8. Query By Example .....	60
11.9. Variables .....	60
11.10. Unbound Variables .....	61
11.11. Unique .....	63
11.12. Result Range and Ordering .....	63
11.13. Projection .....	64
11.14. Projection Field Traversal .....	64
11.15. Projection Variables .....	65
11.16. Distinct Projection .....	65
11.17. Count .....	66
11.18. Min, Max, Avg .....	67
11.19. Grouping .....	67
11.20. Populating a JavaBean .....	69
11.21. Result Aliases .....	69
11.22. Taking Advantage of the Default Result String .....	70
11.23. Populating a Map .....	70
11.24. Defining a Named Query .....	73
11.25. Ordering, Range, Variables, and Parameters .....	74
11.26. Single-String Named Query .....	74
11.27. Aggregates and Projections .....	74
11.28. Executing Named Queries .....	75
12.1. Creating a SQL Query .....	76
12.2. Retrieving Persistent Objects .....	77
12.3. SQL Query Parameters .....	78
12.4. Column Projection .....	78
12.5. Result Class .....	79
12.6. Named SQL Queries .....	79
4.1. Binding a PersistenceManagerFactory into JNDI via a WebLogic Startup Class .....	108
4.2. Looking up the PersistenceManagerFactory in JNDI .....	108
1.1. Issuing a query against a Date field .....	123
2.1. Code Formatting with the Application Id Tool .....	140
3.1. Standard Kodo Log Configuration .....	162

3.2. Standard Kodo Log Configuration + All SQL Statements .....	163
3.3. Logging to a File .....	163
3.4. Standard Log4J Logging .....	163
3.5. JDK 1.4 Log Properties .....	164
3.6. Custom Logging Class .....	165
4.1. Properties File for the Kodo JDO DataSource .....	168
4.2. Properties File for a Third-Party DataSource .....	168
4.3. Specifying a DBDictionary .....	176
4.4. Specifying a Transaction Isolation .....	176
4.5. Specifying the Join Syntax Default .....	177
4.6. Specifying the Join Syntax at Runtime .....	177
4.7. Specifying Connection Usage Defaults .....	179
4.8. Specifying Connection Usage at Runtime .....	179
4.9. Obtaining a JDBC Connection from the PersistenceManager .....	179
4.10. Obtaining a JDBC Connection from the DataSource .....	180
4.11. Specifying Result Set Defaults .....	181
4.12. Specifying Result Set Behavior at Runtime .....	181
4.13. Using Random Access Query Results in a Portable Fashion .....	182
4.14. Enabling SQL Statement Ordering .....	182
5.1. Using the Application Identity Tool .....	184
5.2. Sequence Factory Configuration .....	186
5.3. Accessing the Sequence Factory .....	187
5.4. Auto-Increment Metadata .....	188
5.5. Sequence-Assigned Metadata .....	188
5.6. Enabling Managed Inverses .....	189
5.7. Log Inconsistencies, Including LRS Fields .....	189
5.8. Using Initial Field Values .....	190
5.9. Using a Large Result Set Iterator .....	190
5.10. Marking a Large Result Set Field .....	191
5.11. Configuring the Proxy Manager .....	192
5.12. Using the Kodo JDO Enhancer .....	192
5.13. Using the Schema Generator .....	193
5.14. Using the Reverse Mapping Tool .....	193
5.15. Using the Mapping Tool .....	195
5.16. Customizing Reverse Mapping with Properties .....	195
6.1. Using the MetaDataTool .....	197
7.1. Using the Mapping Tool .....	209
7.2. Refreshing Mappings and the Relational Schema .....	211
7.3. Dropping Mappings .....	211
7.4. Validating Mappings .....	211
7.5. Updating the Schema Based on Mapping Data .....	212
7.6. Modifying Default Mappings .....	212
7.7. Reverting Mapping Data .....	212
7.8. Refresh Mappings and Create DDL .....	212
7.9. Refresh Mappings and Create DDL to Update Database .....	213
7.10. Create DDL for Current Mappings .....	213
7.11. Create DDL to Update Database for Current Mappings .....	213
7.12. Modifying Difficult-to-Access Mapping Data .....	215
7.13. Switching Mapping Factories .....	215
7.14. Basic Structure of Mapping Documents .....	216
7.15. Complete Mapping Document .....	217
7.16. Using a Base Mapping .....	220
7.17. Using a Flat Mapping .....	221
7.18. Using a Vertical Mapping .....	223
7.19. Using a Horizontal Mapping .....	225
7.20. Using a Horizontal Mapping with Application Identity hierarchy .....	226
7.21. Using a Version Number Indicator .....	229
7.22. Using a Version Date Indicator .....	230
7.23. Using a State Image Indicator .....	231

7.24. Using an In-Class-Name Indicator .....	233
7.25. Using a Metadata-Value Indicator .....	234
7.26. Using a Subclass-Join Indicator .....	235
7.27. Using a Value Mapping .....	237
7.28. Using a Value Mapping in a Separate Table .....	238
7.29. Using a Blob Mapping .....	239
7.30. Using a Clob Mapping .....	240
7.31. Using a Byte array Mapping .....	242
7.32. Using a One-to-One Mapping .....	243
7.33. Using a Two-Sided One-to-One Mapping .....	244
7.34. Using a One-to-One Mapping With Inverse Columns .....	245
7.35. Using a PC One-to-One Mapping .....	247
7.36. Using an Embedded One-to-One Mapping .....	248
7.37. Using an Enum Mapping .....	250
7.38. Using a Collection Mapping .....	252
7.39. Using a Many-to-Many Mapping .....	253
7.40. Using a Two-Sided Many-to-Many Mapping .....	254
7.41. Using a One-to-Many Mapping .....	257
7.42. Using a One-Sided One-to-Many Mapping .....	258
7.43. Using a PC Collection Mapping .....	259
7.44. Using a Map Mapping .....	260
7.45. Using an N-to-Many Map Mapping .....	262
7.46. Using a Many-to-N Map Mapping .....	263
7.47. Using a Many-to-Many Map Mapping .....	265
7.48. Using a PC Map Mapping .....	266
7.49. Using an N-to-PC Map Mapping .....	268
7.50. Using a PC-to-N Map Mapping .....	269
7.51. Using a PC-to-Many Map Mapping .....	271
7.52. Using a Many-to-PC Map Mapping .....	272
7.53. Using Externalization .....	275
7.54. Querying Externalization Fields .....	276
7.55. Using External Values .....	277
8.1. Using the Schema Generator .....	279
8.2. Schema Creation .....	281
8.3. SQL Scripting .....	282
8.4. Schema Drop .....	282
8.5. Basic Schema .....	283
8.6. Full Schema .....	283
8.7. Connecting to the Database .....	284
8.8. Examining the Tutorial Schema .....	284
8.9. Issuing SQL Against the Database .....	285
9.1. Specifying the PersistenceManagerFactory .....	286
9.2. Using the JDOHelper .....	286
9.3. Using the KodoHelper .....	286
10.1. Basic JDOQL Extension .....	291
10.2. Chaining JDOQL Extensions .....	291
10.3. Setting Default Lock Levels .....	293
10.4. Setting Runtime Lock Levels .....	293
10.5. Using lockPersistent() .....	294
10.6. Disabling Locking .....	294
10.7. Custom Logging Orphaned Keys .....	296
11.1. Detaching and Attaching a Single Instance .....	299
11.2. Using Custom Fetch Groups for Detach .....	300
11.3. Configuring a Standalone Persistence Manager Server .....	304
11.4. Starting a Standalone Persistence Manager Server .....	304
11.5. Client Configuration .....	307
11.6. HTTP Client Configuration .....	307
11.7. Enabling Compression with the TCP Transport .....	307
11.8. Enabling Compression with the HTTP Transport .....	308

12.1. Accessing the MBeanServer .....	317
13.1. Configuring Transaction Manager Integration .....	321
13.2. XA Configuration .....	322
13.3. Comparison to Subquery .....	323
13.4. Correlated Subquery .....	323
13.5. Subquery Contains .....	323
13.6. Subquery Empty .....	324
14.1. Configuring SQL Batching .....	326
14.2. Setting the Default Eager Fetch Mode .....	327
14.3. Setting the Eager Fetch Mode at Runtime .....	328
14.4. Specifying a DataCache Timeout .....	330
14.5. Specifying a Non-Default DataCache .....	331
14.6. Configuring and Acquiring a Named DataCache .....	331
14.7. Pinning an Object into the DataCache .....	331
14.8. Unpinning an Object from the DataCache .....	331
14.9. Evicting an Object from the DataCache .....	332
14.10. Data Cache Eviction Through the Persistence Manager .....	332
14.11. Setting the Size of the Query Cache .....	332
14.12. Disabling the Query Cache .....	333
14.13. Notifying the Query Cache of Altered Classes .....	333
14.14. Dropping or Pinning Query Results .....	333
14.15. Disabling and Enabling Query Caching .....	334
14.16. Query Replaces Extent .....	335
14.17. JMS Remote Commit Provider Configuration .....	336
14.18. TCP Remote Commit Provider Configuration .....	337
14.19. Custom Fetch Group Meta-Data .....	338
14.20. Adding a Fetch Group to a Query .....	339
14.21. Adding a Single Field to a PersistenceManager .....	339
14.22. Lock Group Meta-Data .....	340
14.23. Lock Group Meta-Data .....	341
14.24. Using Lock Groups with version-number Indicator .....	342
15.1. Using the <config> Ant Tag .....	346
15.2. Using the Properties Attribute of the <config> Tag .....	347
15.3. Using the PropertiesFile Attribute of the <config> Tag .....	347
15.4. Using the <classpath> Ant Tag .....	347
15.5. Using the <codeformat> Ant Tag .....	347
15.6. Invoking the JDO Enhancer from Ant .....	348
15.7. Invoking the Application Identity Tool from Ant .....	348
15.8. Invoking the JDO Metadata Tool from Ant .....	349
15.9. Invoking the Mapping Tool from Ant .....	349
15.10. Invoking the Reverse Mapping Tool from Ant .....	350
15.11. Invoking the Schema Tool from Ant .....	350
15.12. Invoking the Schema Generator from Ant .....	350
15.13. Commenting for XDoclet .....	351
15.14. Invoking XDoclet with Ant .....	354
16.1. Explicitly Closing Resources .....	364
16.2. Disabling the Class Indicator .....	365
16.3. Appropriate use of JDOQL parameters .....	366
16.4. Inappropriate use of JDOQL parameters .....	366
B.1. Example properties for Derby .....	404
B.2. Example properties for Interbase .....	405
B.3. Example properties for JDataStore .....	405
B.4. Example properties for IBM DB2 .....	405
B.5. Example properties for Empress .....	406
B.6. Example properties for Hypersonic .....	406
B.7. Example properties for Firebird .....	407
B.8. Example properties for Informix Dynamic Server .....	407
B.9. Example properties for InterSystems Cache .....	407
B.10. Example properties for Microsoft Access .....	408

B.11. Example properties for Microsoft SQLServer .....	408
B.12. Example properties for Microsoft FoxPro .....	409
B.13. Example properties for MySQL .....	409
B.14. Example properties for Oracle .....	410
B.15. Example properties for Pointbase .....	410
B.16. Example properties for PostgreSQL .....	411
B.17. Example properties for Sybase .....	411
D.1. Using the Kodo 2 Migrator .....	424
D.2. Invoking the Kodo 2 Migrator from Ant .....	424
D.3. Using the Kodo 2 Properties Tool .....	425
D.4. Invoking the Kodo 2 Properties Tool from Ant .....	425
F.1. Tangosol Cache Configuration .....	428
F.2. GemFire Cache Configuration .....	428

---

# Part I. Introduction

---

---

# Table of Contents

- 1. SolarMetric Kodo JDO ..... 3
  - 1.1. About This Document ..... 3
- 2. SolarMetric Kodo JDO Installation ..... 4
  - 2.1. Overview ..... 4
  - 2.2. Updates ..... 4
  - 2.3. Key Files in the download ..... 4
  - 2.4. Quick Start ..... 4
  - 2.5. Upgrading from Kodo 2 ..... 5
  - 2.6. Requirements ..... 5
  - 2.7. Terminology ..... 5
  - 2.8. Windows Installation (installer) ..... 5
  - 2.9. Windows Installation (no installer -- zip file) ..... 5
  - 2.10. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation ..... 6
  - 2.11. Borland JBuilder ..... 6
  - 2.12. Installing Kodo into JBuilder ..... 6
  - 2.13. Common installation problems ..... 6
  - 2.14. Resources ..... 7
  - 2.15. Sales Inquiries ..... 7

---

# Chapter 1. SolarMetric Kodo JDO

Kodo JDO is SolarMetric's implementation of Sun's Java Data Objects (JDO) specification for the transparent persistence of Java objects. This document provides an overview of the JDO standard and technical details on the use of Kodo JDO.

## 1.1. About This Document

---

This document is intended for Kodo JDO users. It is divided into several parts:

- The **JDO Overview** describes the fundamentals of the JDO specification. If you are new to JDO, you should consider this section required reading before moving forward. If you are already familiar with JDO basics, you can skip this section.
- In the **Kodo JDO Tutorials** you will develop simple persistent applications using Kodo. Through the tutorials' hands-on approach, you will become comfortable with the core tools and development processes under Kodo JDO.
- The **Kodo JDO Reference Guide** contains detailed documentation on all aspects of Kodo JDO. Browse through this guide to familiarize yourself with the many advanced features and customizations Kodo provides. Later, you can use the guide when you need details on a specific aspect of Kodo JDO.
- Finally, the **Kodo Development Workbench Guide** describes Kodo's standalone GUI mapping tool. The guide begins with a tutorial on how to get started using Kodo Development Workbench and concludes with a reference on all of its features.

---

# Chapter 2. SolarMetric Kodo JDO Installation

## 2.1. Overview

---

This download includes the commercially available release of Kodo JDO version 3.3.5. Kodo JDO is an implementation of the Java Data Objects standard for relational databases. Version 3.3.5 supports the candidate release of the JDO 1.0.2 specification. A copy of this specification is provided in the documentation of Kodo JDO version 3.3.5.

You just downloaded one of the following:

- `kodo-jdo-3.3.5.tar.gz` - Kodo JDO version 3.3.5 for Unix or Mac OS X.
- `kodo-jdo-3.3.5.exe` - Kodo JDO version 3.3.5 for Microsoft Windows NT and Windows 2000. Includes Windows Installer.
- `kodo-jdo-3.3.5.zip` - Kodo JDO version 3.3.5 for Microsoft Windows NT and Windows 2000.

## 2.2. Updates

---

Please check the SolarMetric web site to download the latest version of Kodo JDO (registration is required to download evaluation copies). The SolarMetric Kodo JDO pages are located at [http://www.solarmetric.com/Software/Kodo\\_JDO/](http://www.solarmetric.com/Software/Kodo_JDO/).

## 2.3. Key Files in the download

---

1. `README.txt` - This file that you are currently reading.
2. `EVALUATION-LICENSE.txt` - A copy of our evaluation license. You must accept this evaluation license prior to using the software.
3. `lib/serp-license.txt` - License agreement for use of Serp libraries.
4. `lib/hypersonic-license.txt` - License agreement to use Hypersonic database.
5. `lib/jakarta-commons-license.txt` - License agreement to use Jakarta Commons libraries.

## 2.4. Quick Start

---

Please choose the appropriate file for your needs and platform.

Follow the instructions in the appropriate installation section below. Note that documentation can be found in the `docs/index.html` file.

If you just downloaded this package as an evaluation, you will receive a license key in the mail shortly. You must copy the license key string from the email and paste it into the `kodo.properties` file in the same directory as each releases `README.txt` file

If you received this package on a CD, you will need to download a valid license key from <http://www.solarmetric.com/Software/Evaluate>. While obtaining your license key, we highly recommend that you download the latest version of Kodo JDO.

---

## 2.5. Upgrading from Kodo 2

---

There are significant differences to properties, mappings, and internal APIs between previous versions of Kodo and Kodo 3. For information about how to migrate from Kodo 2 to Kodo 3, see the migration guide in Appendix D of the Kodo JDO documentation.

## 2.6. Requirements

---

- A valid Kodo JDO license key. Evaluation keys are available at <http://www.solarmetric.com/Software/Evaluate>. To purchase a key, go to <http://www.solarmetric.com/Software/Purchase>.
- JDK 1.2 or greater
- A relational database with JDBC driver, such as Oracle, IBM DB2, Microsoft SQLServer, Sybase, Pointbase, MySQL, PostgreSQL, Hypersonic SQL, or InstantDB. This installation is bundled with Hypersonic, which requires no installation and minimal configuration.

## 2.7. Terminology

---

`JDOHOME`  
the JDO installation directory. This readme is located in `JDOHOME`.

`JDKHOME`  
the JDK installation directory. This is where your Java installation is located.

## 2.8. Windows Installation (installer)

---

1. Double click on the executable.

## 2.9. Windows Installation (no installer -- zip file)

---

1. Read and agree to the license agreement and any third party license agreements.
  2. Edit `JDOHOME/bin/jdocmd.bat` and `JDOHOME/bin/jdocommand.bat` so that `JDODIR` and `JDKHOME` are set correctly.
  3. Run `JDOHOME/bin/jdocmd.bat` or `JDOHOME/bin/jdocommand.bat` (the former relies on 'cmd', the command shell for NT and 2000; the latter uses 'command', the command shell for 95, 98, and ME). All tutorial commands should be executed from this shell.
  4. Open `JDOHOME/docs/index.html` and navigate to the **Kodo JDO Tutorial (part III, chapter 1)**.
  5. Change to the `JDOHOME/tutorial` directory.
  6. Start the tutorial.
-

## 2.10. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation

---

1. Open a shell.
2. Ensure that your CLASSPATH environment variable contains the base Java runtime package (\$JDKHOME/jre/lib/rt.jar for JDK 1.2 or higher).
3. Change to the JDOHOME directory.
4. Type `'chmod a+x bin/*'`. This will give you execute permissions on all packaged shell scripts.
5. Type `'source bin/envsetup'` (On Windows with Cygwin, type `'source bin/cygsetup'`). This will modify your CLASSPATH and PATH environment variables to add in the libraries in JDOHOME/lib and the executables in JDOHOME/bin.
6. Open JDOHOME/docs/index.html and navigate to the [Kodo JDO Tutorial \(part III, chapter 1\)](#).
7. Change to the JDOHOME/tutorial directory.
8. Start the tutorial.

## 2.11. 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.

## 2.12. Installing Kodo into JBuilder

---

To install Kodo support in JBuilder 7 and higher, 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:\JBuilder\, then you would copy all the .jar files from C:\development\kodo\lib\ to C:\JBuilder\lib\ext\, and then copy the C:\development\kodo\lib\KodoJDO.library file to C:\JBuilder\lib\.

Additionally, you must enter your Kodo license key into the Kodo configuration panel within JBuilder. You can open the configuration panel by clicking on the "K" icon in the toolbar. The information entered in this configuration panel is used by the Kodo development tools integrated into JBuilder, such as the Kodo enhancer and the mapping tools.

To validate the installation, you should start (or restart) JBuilder 7 or higher. 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 manual file copying or any other additional steps.

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

## 2.13. Common installation problems

---

*Problem:* Shell error when running '`javac *.java`' or when using '`java`'

*Solution:* Ensure that `java` and `javac` are installed and in your path. To verify that they are installed, type '`java`' on a line by itself. If this returns a usage statement, then `java` is installed and in your path. Repeat with '`javac`' instead of '`java`' to see if `javac` is installed correctly. If these tests fail, install JDK 1.2 or greater. Also, please make sure that `jdcmd.bat` sets up your path properly.

*Problem:* When running '`jdoc`', you get a `NoClassDefFoundError` with a message like the following:  
`java.lang.NoClassDefFoundError: Animal (wrong name: tutorial/Animal)`

*Solution:* This often means that you are invoking `jdoc` from the directory that contains the class `Animal`, and `'.'` is in your classpath. Fixing your classpath or running `jdoc` from a different directory should solve this problem.

*Problem:* A 'sealing violation' occurs when running the enhancer.

*Solution:* This indicates that some other library in your `CLASSPATH` is conflicting with the jars packaged with this distribution.

## 2.14. Resources

---

If you have any technical questions while evaluating or installing Kodo JDO, please send an email to `<jdosupport@solarmetric.com>`.

The web site (<http://www.solarmetric.com>) has resources for any developer working with Kodo JDO including frequently asked questions, our bug tracking system, SolarMetric's newsgroups, access to technical support, and links to a variety of technical articles and third party tutorials.

## 2.15. Sales Inquiries

---

`<sales@solarmetric.com>`

+1 202-595-2064 x2

---

## **Part II. Java Data Objects**

---

---

# Table of Contents

1. Introduction .....	11
1.1. Intended Audience .....	11
1.2. Transparent Persistence .....	11
2. Why JDO? .....	12
3. JDO Architecture .....	14
3.1. JDO Exceptions .....	15
4. PersistenceCapable .....	16
4.1. JDO Enhancer .....	16
4.2. Persistence-Capable vs. Persistence-Aware .....	17
4.3. Restrictions on Persistent Classes .....	17
4.3.1. Default or No-Arg Constructor .....	17
4.3.2. Inheritance .....	17
4.3.3. Persistent Fields .....	18
4.3.4. Conclusions .....	20
4.4. InstanceCallbacks .....	20
4.5. JDO Identity .....	21
4.5.1. Datastore Identity .....	22
4.5.2. Application Identity .....	23
4.5.2.1. Application Identity Hierarchies .....	25
4.5.3. Single Field Identity .....	26
4.6. Conclusions .....	26
5. Metadata .....	27
5.1. Metadata DTD .....	27
5.2. Metadata Placement .....	31
6. JDOHelper .....	33
6.1. Persistence-Capable Operations .....	33
6.2. Lifecycle Operations .....	33
6.3. PersistenceManagerFactory Construction .....	36
7. PersistenceManagerFactory .....	37
7.1. Obtaining a PersistenceManagerFactory .....	37
7.2. PersistenceManagerFactory Properties .....	37
7.2.1. Connection Configuration .....	38
7.2.2. PersistenceManager and Transaction Defaults .....	39
7.3. Obtaining PersistenceManagers .....	40
7.4. Properties and Supported Options .....	41
8. PersistenceManager .....	43
8.1. User Object Association .....	44
8.2. Configuration Properties .....	44
8.3. Transaction Association .....	44
8.4. Persistence-Capable Lifecycle Management .....	44
8.5. Lifecycle Examples .....	46
8.6. JDO Identity Management .....	47
8.7. Extent Factory .....	48
8.8. Query Factory .....	48
8.9. Closing .....	49
9. Transaction .....	50
9.1. Transaction Types .....	50
9.2. The JDO Transaction Interface .....	51
10. Extent .....	53
11. Query .....	54
11.1. Object Filtering .....	54
11.2. JDOQL .....	56
11.3. Advanced Object Filtering .....	58

11.4. Compiling and Executing Queries .....	61
11.5. Limits and Ordering .....	62
11.6. Projections .....	64
11.7. Aggregates .....	66
11.8. Result Class .....	68
11.8.1. JavaBean Result Class .....	69
11.8.2. Generic Result Class .....	70
11.9. Single-String JDOQL .....	70
11.10. Named Queries .....	72
11.10.1. Named Query DTD .....	72
11.10.2. Named Query Examples .....	73
11.11. Conclusion .....	75
12. SQL Queries .....	76
12.1. Creating SQL Queries .....	76
12.2. Retrieving Persistent Objects with SQL .....	77
12.3. SQL Projections .....	78
12.4. Named SQL Queries .....	79
12.5. Conclusion .....	80
13. Conclusion .....	81

---

# 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). 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, 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, as illustrated by the table below.

**Table 2.1. Persistence Mechanisms**

<b>Supports:</b>	<b>Serialization</b>	<b>JDBC</b>	<b>O-R Tool</b>	<b>Object DB</b>	<b>EJB</b>	<b>JDO</b>
Java Objects	<b>Yes</b>	No	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
Advanced OO Concepts	<b>Yes</b>	No	<b>Yes</b>	<b>Yes</b>	No	<b>Yes</b>
Transactional Integrity	No	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
Concurrency	No	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
Large Data Sets	No	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
Existing Schema	No	<b>Yes</b>	<b>Yes</b>	No	<b>Yes</b>	<b>Yes</b>
Queries	No	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
Strict Standards / Portability	<b>Yes</b>	No	No	No	<b>Yes</b>	<b>Yes</b>
Simplicity	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	No	<b>Yes</b>

- *Serialization* is Java's built-in mechanism for transforming an object graph into a series of bytes, which can then be sent over the network or stored in a file. Serialization is very easy to use, but it is also very limited. It must store and retrieve the entire object graph at once, making it unsuitable for dealing with large amounts of data. It cannot undo changes that are made to objects if an error occurs while updating information, making it unsuitable for applications that require strict data integrity. Multiple threads or programs cannot read and write the same serialized data concurrently without conflicting with each other. It provides no query capabilities. All these factors make serialization useless for all but the most trivial persistence needs.
- Many developers use the *Java Database Connectivity* (JDBC) APIs to manipulate persistent data in relational databases. JDBC overcomes most of the shortcomings of serialization: it can handle large amounts of data, has mechanisms to ensure data integrity, supports concurrent access to information, and has a sophisticated query language in SQL. Unfortunately, JDBC does not duplicate serialization's ease of use. The relational paradigm used by JDBC was not designed for storing objects, and therefore forces you to either abandon object-oriented programming for the portions of your code that deal with persistent data, or to find a way of mapping object-oriented concepts like inheritance to relational databases yourself.
- Several software companies created frameworks to perform the mapping between objects and relational database tables for you. These *object-relational mapping* products allow you to focus on the object model and not concern yourself with the mismatch between the object-oriented and relational paradigms. Unfortunately, each object-relational mapping product has its own set of APIs. Your code becomes tied to the proprietary interfaces of a single vendor. If the vendor raises prices or fails to fix show-stopping bugs, you cannot switch to another product without rewriting all of your persistence code. This is referred to as vendor lock-in.
- Rather than map objects to relational databases, some software companies developed a new form of database designed specifically to store objects. These *object databases* are often much easier to use than object-relational mapping software. The Object Database Management Group (ODMG) was formed to create a standard API for accessing object databases; few object database vendors, however, comply with the ODMG's recommendations. Thus, vendor lock-in plagues object databases as well. Many companies are also hesitant to switch from tried-and-true relational systems to the relatively new object database technology. Fewer data-analysis tools are available for object database systems, and there are vast quantities of data

already stored in older relational databases. For all of these reasons and more, object databases have not caught on as well as their creators hoped.

- The Enterprise Edition of the Java platform introduced entity Enterprise Java Beans (EJBs). Entity EJBs are components that represent persistent information in a data store. Like object-relational mapping solutions, entity EJBs provide an object-oriented view of persistent data. Unlike object-relational software, however, entity EJBs are not limited to relational databases; the persistent information they represent may come from an Enterprise Information System (EIS) or other storage device. Also, EJBs use a strict standard, making them portable across vendors. Unfortunately, the EJB standard is somewhat limited in the object-oriented concepts it can represent. Advanced features like inheritance, polymorphism, and complex relations are absent. Additionally, EJBs are difficult to code, and they require heavyweight and often expensive application servers to run. EJBs, especially session and message-driven beans, do have other advantages, however, and so the JDO specification details how JDO can integrate with them.

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

### Note

By default, Kodo JDO stores objects in relational databases using JDBC. It can be customized for use with other data stores.

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 below illustrates the relationships between the primary components of the JDO architecture.

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

The example below illustrates how the JDO interfaces interact to execute a query and update persistent objects.

## *Example 3.1. Interaction of JDO Interfaces*

```
// get a persistence manager factory using the jdo helper
PersistenceManagerFactory factory =
    JDOHelper.getPersistenceManagerFactory (System.getProperties ());

// get a persistence manager from the factory
PersistenceManager pm = factory.getPersistenceManager ();

// updates take place within transactions
Transaction tx = pm.currentTransaction ();
tx.begin ();

// query for all employees who work in our research division
// and put in over 40 hours a week average
Extent extent = pm.getExtent (Employee.class, false);
Query query = pm.newQuery ();
query.setCandidates (extent);
query.setFilter ("division.name == 'Research' && avgHours > 40");
Collection results = (Collection) query.execute ();

// give all those hard-working employees a raise
Employee emp;
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    emp = (Employee) itr.next ();
    emp.setSalary (emp.getSalary () * 1.1);
}

// commit the updates and free resources
```

```
tx.commit ();  
pm.close ();  
factory.close ();
```

The remainder of this document explores the JDO interfaces in detail. We present them in roughly the order that you will use them as you develop your application.

## 3.1. JDO Exceptions

---

The diagram below depicts 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

In JDO, 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 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;

    private Magazine ()
    {
    }

    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;
    }

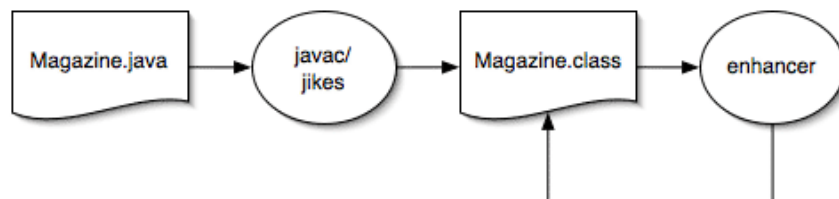
    public void addArticle (Article article)
    {
        articles.add (article);
    }

    // rest of methods omitted
}
```

## 4.1. JDO Enhancer

---

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. Though some vendors may use source enhancers that modify your Java code, enhancers generally operate on `.class` files. They post-process the bytecode generated by your Java compiler, adding the necessary fields and methods to implement the `PersistenceCapable` interface. JDO's bytecode modification perfectly preserves the line numbers in stack traces and is compatible with Java debuggers, so enhancement does not affect debugging.



The diagram above illustrates the compilation of a persistent class. JDO implementations typically include an Ant enhancer task so that you can make enhancement an automatic part of your build process.

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 the class implement the `PersistenceCapable` interface.

Generally, it is best to keep all of your persistent fields private, or protected but only accessed by persistent 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. Default or No-Arg Constructor

---

The JDO specification requires that all persistence-capable classes must have a no-arg constructor. This constructor may be private, if desired. Because the compiler automatically creates a default no-arg constructor when no other constructor is defined, only classes that define constructors must also include a no-arg constructor.

#### Note

Kodo JDO's enhancer will automatically add a protected no-arg constructor to your class when required. Therefore, this restriction does not apply under Kodo.

### 4.3.2. 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 either use the same identity class, or else they must each declare that they use separate identity classes whose inheritance hierarchy and whose modifiers (such as whether they are abstract) exactly mirror the inheritance hierarchy of the persistent class hierarchy. We will cover JDO identity shortly.

### 4.3.3. 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.lang.Number`
- `java.util.Locale`

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

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

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

#### Note

Many JDO implementations support more than just the `HashSet` and `Date` mutable types. Kodo JDO supports the following:

- `java.util.Date`
- `java.util.List`

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

Kodo JDO allows you to plug in direct support for additional mutable types as well.

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 either reset the array field, or 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");  
// or: child.setObjectArray (child.getObjectArray ());  
}
```

As the parent-child 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.

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 these fields are so general, 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 user-defined persistent objects as elements of any of the supported collection types. It also supports user-defined persistent objects as keys, values, or both in any supported map type.

Kodo JDO supports persistent `java.lang.Object` fields by serializing the field value and storing it as a sequence of bytes. It supports persistent interface fields by storing the unique id value of the object stored in the field, then re-fetching the corresponding object when the field is loaded. Collections and maps where the element/key/value type is `java.lang.Object` or an interface are fully supported as well.

## 4.3.4. 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 explores 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 primitive 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. An example of this is presented below.

- `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. See the example below.

- 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.

### ***Example 4.3. Using the InstanceCallbacks Interface***

```
/**
 * Example demonstrating the use of the InstanceCallbacks interface to
 * persist a java.net.InetAddress and implement a privately-owned relation.
 */
public class Host
    implements InstanceCallbacks
{
    // the InetAddress field cannot be persisted directly by JDO, so we
    // use the jdoPostLoad and jdoPreStore methods below to persist it
    // indirectly through its host name string
    private transient InetAddress address;    // non-persistent
    private String      hostName;    // persistent

    // set of devices attached to this host
    private Set devices = new HashSet ();

    // setters, getters, and business logic omitted

    public void jdoPostLoad ()
    {
        // form the InetAddress using the persistent host name
        try
        {
            address = InetAddress.getByName (hostName);
        }
        catch (IOException ioe)
        {
            throw new JDOException ("Invalid host name: " + hostName, ioe);
        }
    }

    public void jdoPreStore ()
    {
        // store the host name information based on the InetAddress values
        hostName = address.getHostName ();
    }

    public void jdoPreDelete ()
    {
        // delete all related devices when this host is deleted
        JDOHelper.getPersistenceManager (this).deletePersistentAll (devices);
    }

    public void jdoPreClear ()
    {
    }
}
```

## **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 numeric 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.

The JDO identity of each persistent instance is encapsulated in its *JDO identity object*. You can obtain the JDO identity object for 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.

### Example 4.4. JDO Identity Objects

```
/**
 * This method tests whether the given persistent objects represent the
 * same data store record. It returns false if either argument is not
 * a persistent object.
 */
public boolean persistentEquals (Object obj1, Object obj2)
{
    Object jdoId1 = JDOHelper.getObjectId (obj1);
    Object jdoId2 = JDOHelper.getObjectId (obj2);
    return jdoId1 != null && jdoId1.equals (jdoId2);
}
```

If you are dealing with a single **PersistenceManager**, then there is an even easier way to test whether two persistent object references represent the same state in the data store: 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 numeric 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. All persistent classes in an inheritance tree must use the same form of JDO identity.

#### 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 class is used for JDO identity objects, or what data is used to create identity values. The only requirement placed on JDO vendors implementing datastore identity is that the class they use for JDO identity objects meets the following criteria:

- The class must be public.
- The class must be serializable.
- All non-static fields of the class must be public and serializable.
- The class must have a public no-args constructor.

- The class must have a public `String` constructor. It must override the `toString` method to return a string that can be used by this constructor to create a new JDO identity object that compares equal to the instance the string was obtained from.

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

### Note

Kodo JDO allows you to customize the manner in which datastore identity values are generated.

## 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 field values must be unique among all other objects of the same type.

When using application identity, the `equals` and `hashCode` methods of the persistence-capable class must depend on all of the primary key fields.

### Note

Kodo does not depend upon this behavior. However, some JDO implementations do, so it should be implemented if portability is a concern.

If you have one one primary key field, you can use *single field identity* to simplify working with application identity (see [Section 4.5.3, “Single Field Identity” \[26\]](#)). Otherwise, it is up to you to supply the class used for JDO identity objects. Your 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, or else each class must have its own application identity class whose inheritance hierarchy mirrors the inheritance hierarchy of the owning persistent classes (see [Section 4.5.2.1, “Application Identity Hierarchies” \[25\]](#)).
- Primary key fields must be primitives, primitive wrappers, or `Strings`. Notably, other persistent instances can *not* be used as primary key fields.

### Note

For legacy schemas with binary primary key columns, Kodo also supports using primary key fields of type `byte[]`.

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-cap-

able class with the JVM. This is typically accomplished with a static block in the application identity class code, as the example below illustrates. 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 `appidtool` to automatically generate proper application identity classes based on your primary key fields.

### Example 4.5. Application Identity Class

```
/**
 * Persistent class using application identity.
 */
public class Magazine
{
    private String isbn;    // primary key field
    private String title;  // primary key field

    /**
     * Equality must be implemented in terms of primary key field
     * equality, and must use instanceof rather than comparing
     * classes directly.
     */
    public boolean equals (Object other)
    {
        if (other == this)
            return true;
        if (!(other instanceof Magazine))
            return false;

        Magazine mag = (Magazine) other;
        return ((isbn == null && mag.isbn == null)
            || (isbn != null && isbn.equals (mag.isbn)))
            && ((title == null && mag.title == null)
            || (title != null && title.equals (mag.title)));
    }

    /**
     * Hashcode must also depend on primary key values.
     */
    public int hashCode ()
    {
        return ((isbn == null) ? 0 : isbn.hashCode ())
            + ((title == null) ? 0 : title.hashCode ())
            % Integer.MAX_VALUE;
    }

    // rest of fields and methods omitted

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

        // each primary key field in the Magazine class must have a
        // corresponding public field in the identity class
        public String isbn;
        public String title;

        /**
         * Default constructor requirement.
         */
        public MagazineId ()
        {
        }

        /**
         * String constructor requirement.
         */
        public MagazineId (String str)
        {
            int idx = str.indexOf (':');
            isbn = str.substring (0, idx);
        }
    }
}
```

```

        title = str.substring (idx + 1);
    }

    /**
     * toString must return a string parsable by the string constructor.
     */
    public String toString ()
    {
        return isbn + ":" + title;
    }

    /**
     * Equality must be implemented in terms of primary key field
     * equality, and must use instanceof rather than comparing
     * classes directly (some JDO implementations may subclass JDO
     * identity class).
     */
    public boolean equals (Object other)
    {
        if (other == this)
            return true;
        if (!(other instanceof MagazineId))
            return false;

        MagazineId mi = (MagazineId) other;
        return ((isbn == null && mi.isbn == null)
            || (isbn != null && isbn.equals (mi.isbn)))
            && ((title == null && mi.title == null)
            || (title != null && title.equals (mi.title)));
    }

    /**
     * Hashcode must also depend on primary key values.
     */
    public int hashCode ()
    {
        return ((isbn == null) ? 0 : isbn.hashCode ())
            + ((title == null) ? 0 : title.hashCode ())
            % Integer.MAX_VALUE;
    }
}

```

### 4.5.2.1. Application Identity Hierarchies

An alternative to having a single application identity class for an entire inheritance hierarchy is to have one application identity class per level in the inheritance hierarchy. The requirements for using a hierarchy of application identity classes is as follows:

- The inheritance hierarchy of application identity classes must exactly mirror the hierarchy of the persistent classes that they identify. For example, if there is abstract class `Person`, abstract class `Employee` that extends `Person`, non-abstract class `FullTimeEmployee` that extends `Employee`, and non-abstract `Manager` that extends `FullTimeEmployee`, then the corresponding application identity classes might be an abstract `PersonID` class, and abstract `EmployeeID` that extends `PersonID`, a non-abstract `FullTimeEmployeeID` class that extends `EmployeeID`, and a non-abstract `ManagerID` class that extends `FullTimeEmployeeID`.
- Subclasses in the application identity hierarchy may define additional primary key fields until the hierarchy becomes non-abstract. In the aforementioned example, `PersonID` may define an ID field called `pk1`, `EmployeeID` may define an additional primary key field called `pk2`, and `FullTimeEmployeeID` can further define an additional primary key field called `pk3`. However, `ManagerID` may not define any additional primary key fields, since it is a subclass of a non-abstract class.

#### Note

When defining additional primary key fields in subclasses of application identity classes, Kodo requires that you use the "horizontal" class mapping (see [Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#))

- It is not necessary for each abstract application identity class to declare primary key fields. In the previous example, the abstract `PersonID` and `EmployeeID` classes could declare no primary key fields, and the first concrete subclass `Full-`

TimeEmployeeID could define one or more primary key fields.

### 4.5.3. Single Field Identity

---

Single field identity is a subset of application identity. When you have only one primary key field, you can choose to use one of JDO's built-in single field identity classes instead of coding your own application identity class. The following list enumerates the primary key field types supported by single field identity, and the built-in identity class for each type:

- `byte, java.lang.Byte: javax.jdo.ByteIdentity`
- `char, java.lang.Character: javax.jdo.CharIdentity`
- `int, java.lang.Integer: javax.jdo.IntIdentity`
- `long, java.lang.Long: javax.jdo.LongIdentity`
- `short, java.lang.Short: javax.jdo.ShortIdentity`
- `java.lang.String: javax.jdo.StringIdentity`

#### Note

Single field identity is a planned JDO 2 feature. It is possible that it will change before the JDO 2 specification is finalized. Official JDO 2 jars are not available yet; until they are you must use the Kodo-supplied single field identity classes rather than the official classes above: `kodo.util.ByteIdentity`, `kodo.util.CharIdentity`, `kodo.util.IntIdentity`, `kodo.util.LongIdentity`, `kodo.util.ShortIdentity`, `kodo.util.StringIdentity`.

These Kodo-specific classes will be removed once JDO 2 jars become available.

## 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 default JDO behavior.
3. To provide the JDO implementation with information that it cannot glean from simply reflecting on the persistence-capable class.

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

## 5.1. Metadata DTD

---

```
<!ELEMENT jdo (package)+>

<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>

<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|none) 'datastore'>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>

<!ELEMENT field ((collection|map|array)?, (extension)*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier (persistent|transactional|none) 'persistent'>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value (exception|default|none) 'none'>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>

<!ELEMENT array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>

<!ELEMENT collection (extension)*>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>

<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>

<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

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

### *Example 5.1. Basic Structure of Metadata Documents*

```
<?xml version="1.0"?>
<jdo>
```

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

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

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

### Note

Kodo JDO defines many useful metadata extensions. See the Kodo JDO Reference Guide [chapter on metadata extensions](#) for a full list.

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

Several metadata attributes require you to specify class names. The names you give should follow these 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$inner-class`. For example, `Subscription-Form$LineItem`.

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

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

- `objectid-class`: For application identity, the name of the JDO identity class used by this persistent class. To use single field application identity (see [Section 4.5.3, “Single Field Identity” \[26\]](#)), do not specify this attribute. If your persistent subclass uses the same `objectid-class` as its superclass, you do not have to specify this attribute.
- `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">
    <!-- application identity -->
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      ...
    </class>
    <!-- single field identity -->
    <class name="Article" identity-type="application">
      ...
    </class>
    <!-- default datastore identity -->
    <class name="Author">
      ...
    </class>
    <class name="Address">
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Form">
      ...
    </class>
    <!-- inheritance -->
    <class name="SubscriptionForm" persistence-capable-superclass="Form">
      ...
    </class>
    <!-- static inner class -->
    <class name="SubscriptionForm$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`.

### Note

Kodo JDO allows you to define multiple fetch groups. It also supports eager fetching of related objects when a fetch group is loaded for maximum performance. See [Section 14.5, “Fetch Groups” \[337\]](#) and [Section 14.2, “Eager Fetching” \[326\]](#).

- `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`, array, collection, and map types. All other types default to `false`. Embedded objects do not appear in the `Extent` for their class, and cannot be retrieved by query.

### Note

When you mark a relation to another persistence-capable object as `embedded`, Kodo JDO stores the object in the same table row in the database as the parent object. Kodo even allows recursively embedded objects (a `Company` has an embedded `Address` which has an embedded `PhoneNumber`, for example).

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`, unless the collection field in the object is defined using Java 5 generics, in which case the type information will default to the generic type information.

map elements define four attributes. They are:

- `key-type`: The class of objects used for map keys. Defaults to `java.lang.Object`, or to the Java 5 generic type information for the key if available.
- `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`, or to the Java 5 generic type information for the key if available.
- `embedded-value`: Same as the `embedded-element` element of arrays and collections, but applies to map values.

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

### *Example 5.3. Complete Metadata Document*

```
<?xml version="1.0"?>
<!-- Note that all persistence-capable classes must be listed, but -->
<!-- very few fields need to be specified -->
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      <field name="isbn" primary-key="true"/>
      <field name="title" primary-key="true"/>
      <field name="articles">
        <collection element-type="Article"/>
      </field>
    </class>
    <class name="Article" identity-type="application">
      <field name="id" primary-key="true"/>
      <field name="authors">
        <map key-type="String" value-type="Author"/>
      </field>
    </class>
    <class name="Author">
      <field name="address" embedded="true"/>
    </class>
    <class name="Address"/>
  </package>
  <package name="org.mag.subscribe">
    <class name="Form"/>
    <class name="SubscriptionForm" persistence-capable-superclass="Form">
      <field name="lineItems">
        <collection element-type="SubscriptionForm$LineItem"/>
        <extension vendor-name="kodo" key="inverse-owner" value="form"/>
      </field>
    </class>
    <class name="SubscriptionForm$LineItem"/>
  </package>
</jdo>
```

## 5.2. Metadata Placement

---

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

1. In a resource called `class-name.jdo`, where `class-name` is the name of the class the document applies to, without package name. The resource must be located in the same package as the class.
2. In a resource called `package.jdo`. The resource should be placed in the corresponding package, or in any ancestor 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. They may also contain the metadata for classes in any sub-packages.

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/package.jdo`
- `org/package.jdo`
- `package.jdo`

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

---

# Chapter 6. JDOHelper

```
JDOHelper

static void makeDirty(Object pc, String field)
static Object getObjectId(Object pc)
static PersistenceManager getPersistenceManager(Object pc)

static boolean isDirty(Object pc)
static boolean isTransactional(Object pc)
static boolean isPersistent(Object pc)
static boolean isNew(Object pc)
static boolean isDeleted(Object pc)

static PersistenceManagerFactory getPersistenceManagerFactory(Properties props)
```

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

---

```
public static void makeDirty (Object pc);
public static Object getObjectId (Object pc);
public static PersistenceManager getPersistenceManager (Object pc);
```

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

---

```
public static boolean isDirty (Object pc);
public static boolean isTransactional (Object pc);
public static boolean isPersistent (Object pc);
public static boolean isNew (Object pc);
public static boolean isDeleted (Object pc);
```

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.
- *Persistent-new*. The persistent-new state is reserved for objects that have been made persistent by passing them to `PersistenceManager.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.

On transaction commit, the information in the persistent-new object is inserted into the data store. 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 `PersistenceManager.makePersistent` and then deleted with `PersistenceManager.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.
- *Persistent-dirty*. Persistent objects that have been changed within the current transaction are persistent-dirty. On transaction commit, the data store will be updated to reflect the object's persistent state.
- *Persistent-deleted*. If a persistent object is the parameter of a call to the `PersistenceManager.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.

- *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 reloaded 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.

- *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 stale might sound useless; if they are utilized carefully, however, persistent-nontransactional objects can sometimes offer large performance gains, with little danger of employing outdated 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.

- *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-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.

The following diagram displays the state transitions for persistent objects. Each arrow represents a change from one state to another, and the text next to the arrow indicates the event that triggers change. Method names in purple are methods of the `Transaction` interface. Method names in red are methods of the `PersistenceManager` interface. These interfaces are covered later in this document.

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 the named property, 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.

**Table 6.1. JDOHelper Lifecycle Methods**

	Persistent	Transactional	Dirty	New	Deleted
Transient					
Transient-Clean		X			
Transient-Dirty		X	X		
Persistent-New	X	X	X	X	
Persistent-Nontransactional	X				
Persistent-Clean	X	X			
Persistent-Dirty	X	X	X		
Persistent-Deleted	X	X	X		X
Persistent-	X	X	X	X	X

	Persistent	Transactional	Dirty	New	Deleted
New-Deleted					

## 6.3. PersistenceManagerFactory Construction

```
public static PersistenceManagerFactory getPersistenceManagerFactory (Properties props);
```

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 ();

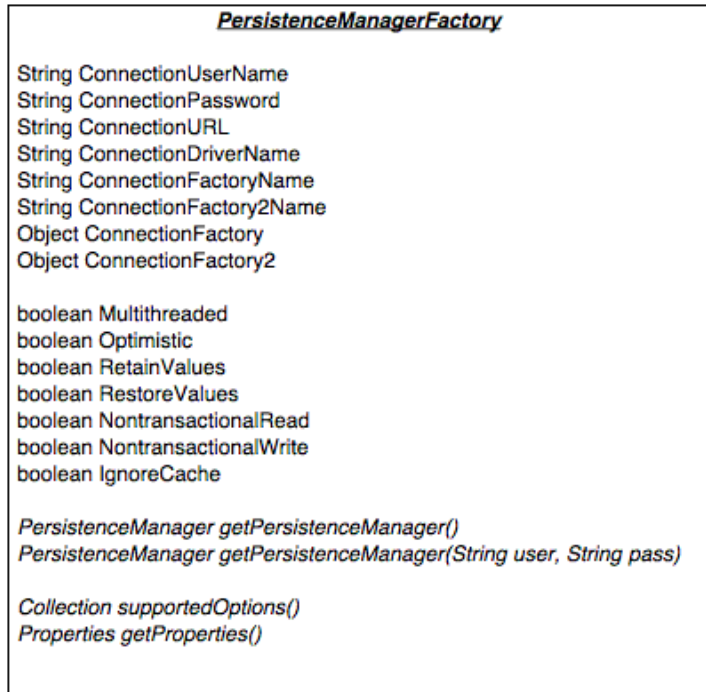
// this property key tells the jdohelper what pmfactory class to instantiate
props.setProperty ("javax.jdo.PersistenceManagerFactoryClass",
    "kodo.jdbc.runtime.JDBCPersistenceManagerFactory");

// 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



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 two functional categories: data store connection configuration and default *PersistenceManager* and Transaction options.

The sections 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 the properties configuration code is displayed along with the method declarations.

---

## 7.2.1. Connection Configuration

---

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

```
public String getConnectionUserName ();
public void setConnectionUserName (String user);
props.setProperty ("javax.jdo.option.ConnectionUserName", user);
```

The user name to specify when connecting to the data store.

```
public String getConnectionPassword ();
public void setConnectionPassword (String pass);
props.setProperty ("javax.jdo.option.ConnectionPassword", pass);
```

The password to specify when connecting to the data store.

```
public String getConnectionURL ();
public void setConnectionURL (String url);
props.setProperty ("javax.jdo.option.ConnectionURL", url);
```

The URL of the data store.

```
public String getConnectionDriverName ();
public void setConnectionDriverName (String driver);
props.setProperty ("javax.jdo.option.ConnectionDriverName", driver);
```

The full class name of the driver to use when interacting with the data store.

```
public String getConnectionFactoryName ();
public void setConnectionFactoryName (String name);
props.setProperty ("javax.jdo.option.ConnectionFactoryName", name);
```

The JNDI location of a connection factory to use to obtain data store connections. This property overrides the other data store connection properties above.

```
public Object getConnectionFactory ();
public void setConnectionFactory (Object factory);
```

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.

## Note

Kodo JDO uses JDBC DataSources as connection factories.

```
public String getConnectionFactory2Name ();
public void setConnectionFactory2Name (String name);
props.setProperty ("javax.jdo.option.ConnectionFactory2Name", name);
```

In a managed environment, connections obtained from the primary connection factory may be automatically enlisted in any global transaction in progress. The JDO implementation might require an additional connection factory that is not configured to participate in global transactions. For example, Kodo's default algorithm for datastore identity generation requires its own non-managed transaction. The JNDI location of this factory can be specified through this property.

```
public Object getConnectionFactory2 ();
public void setConnectionFactory2 (Object factory);
```

The connection factory to use for local transactions. Overrides the `ConnectionFactory2Name` above.

## 7.2.2. PersistenceManager 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`.

```
public boolean getMultithreaded ();
public void setMultithreaded (boolean multithreaded);
props.setProperty ("javax.jdo.option.Multithreaded", multithreaded);
```

Set this property to `true` to indicate that `PersistenceManagers` or the persistence-capable objects they manage will be accessed concurrently by multiple threads in your application. Some JDO implementations might optimize performance by avoiding any synchronization when this property is left `false`.

```
public boolean getOptimistic ();
public void setOptimistic (boolean optimistic);
props.setProperty ("javax.jdo.option.Optimistic", optimistic);
```

Set to `true` to use optimistic transactions by default. The section on **transaction types** discusses optimistic transactions.

```
public boolean getRetainValues ();
public void setRetainValues (boolean retain);
props.setProperty ("javax.jdo.option.RetainValues", retain);
```

If this property is `true`, the fields of persistent objects will not be cleared on transaction commit. Otherwise, persistent fields are

cleared on commit and re-read from the data store the next time they are accessed.

```
public boolean getRestoreValues ();
public void setRestoreValues (boolean restore);
props.setProperty ("javax.jdo.option.RestoreValues", restore);
```

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.

```
public boolean getNontransactionalRead ();
public void setNontransactionalRead (boolean read);
props.setProperty ("javax.jdo.option.NontransactionalRead", read);
```

Specifies whether you can read persistent state outside of a transaction. If this property is `false`, any attempt to iterate an extent, execute a query, or access non-primary key persistent object fields outside of a transaction will result in a `JDOUserException`.

```
public boolean getNontransactionalWrite ();
public void setNontransactionalWrite (boolean write);
props.setProperty ("javax.jdo.option.NontransactionalWrite", write);
```

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`.

Note that changes made outside of a transaction are never flushed to the data store. The changes are discarded as soon as the modified object enters a transaction.

```
public boolean getIgnoreCache ();
public void setIgnoreCache (boolean ignore);
props.setProperty ("javax.jdo.option.IgnoreCache", ignore);
```

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 flush changes to the data store before running queries, or to run transactional queries in memory, both of which can have a negative impact on performance.

### Note

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

---

## 7.3. Obtaining PersistenceManagers

```
public PersistenceManager getPersistenceManager ();
public PersistenceManager getPersistenceManager (String user, String pass);
```

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

---

```
public Properties getProperties ();
public Collection supportedOptions ();
```

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

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

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

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

- `javax.jdo.option.Vector`
- `javax.jdo.option.Map`
- `javax.jdo.option.List`
- `javax.jdo.option.Array`
- `javax.jdo.option.NullCollection`
- `javax.jdo.option.ChangeApplicationIdentity`
- `javax.jdo.query.JDOQL`

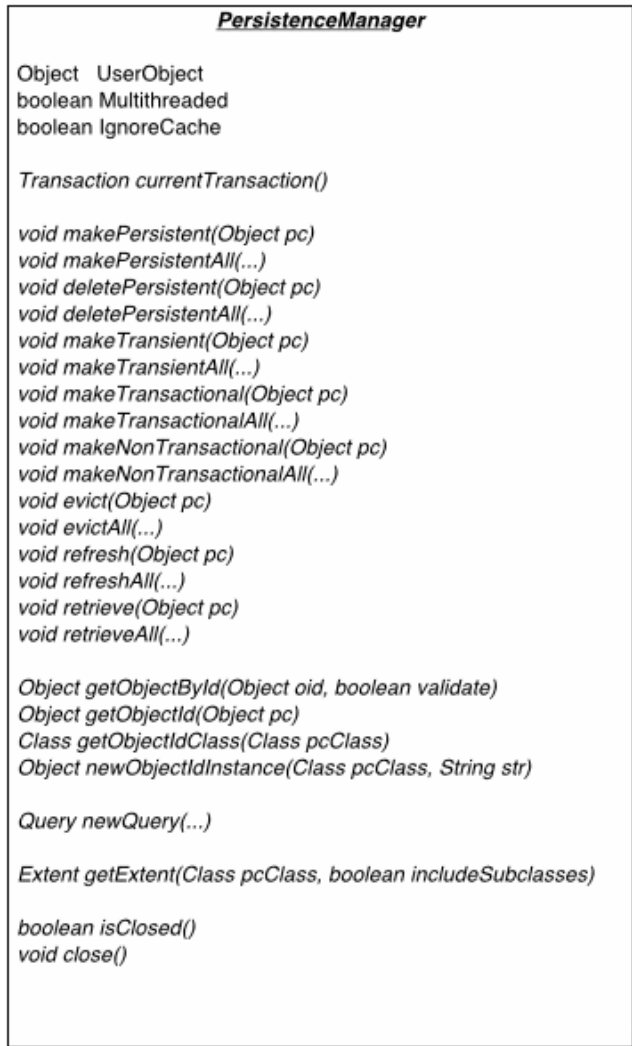
Vendors may include Strings for other query languages they support as well.

### Note

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

---

# Chapter 8. PersistenceManager



The diagram above presents an overview of the most commonly-used methods and properties of the `PersistenceManager` interface. For a complete treatment of the `PersistenceManager` API, see the [Javadoc](#) documentation. Java bean-like properties with "getter" and "setter" methods are listed as field declarations. Methods whose parameter signatures consist of an ellipsis (...) are overloaded to take multiple parameter types.

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

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

- User object association.
- Configuration properties.
- Transaction association.
- Persistence-capable lifecycle management.

- JDO identity management.
- Query factory.
- Extent factory.
- Closing.

## 8.1. User Object Association

---

```
public Object getUserObject ();  
public void setUserObject (Object obj);
```

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

---

```
public boolean getMultithreaded ();  
public void setMultithreaded (boolean threaded);  
public boolean getIgnoreCache ();  
public void setIgnoreCache (boolean ignore);
```

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

---

```
public Transaction currentTransaction ();
```

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 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`.

```
public void makePersistent (Object pc);
public void makePersistentAll (Collection pcs);
public void makePersistentAll (Object[] pcs);
```

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.

```
public void deletePersistent (Object pc);
public void deletePersistentAll (Collection pcs);
public void deletePersistentAll (Object[] pcs);
```

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.

```
public void makeTransient (Object pc);
public void makeTransientAll (Collection pcs);
public void makeTransientAll (Object[] pcs);
```

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.

```
public void makeTransactional (Object pc);
public void makeTransactionalAll (Collection pcs);
public void makeTransactionalAll (Object[] pcs);
```

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.

```
public void makeNontransactional (Object pc);
public void makeNontransactionalAll (Collection pcs);
public void makeNontransactionalAll (Object[] pcs);
```

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`.

```
public void evict (Object pc);
public void evictAll (Collection pcs);
public void evictAll (Object[] pcs);
public void evictAll ();
```

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.

```
public void refresh (Object pc);
public void refreshAll (Collection pcs);
public void refreshAll (Object[] pcs);
public void refreshAll ();
```

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.

```
public void retrieve (Object pc);
public void retrieveAll (Collection pcs);
public void retrieveAll (Object[] pcs);
public void retrieveAll (Collection pcs, boolean dfgOnly);
public void retrieveAll (Object[] pcs, boolean dfgOnly);
```

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. Note, however, that this action is not recursive. That is, if object A has a relation to object B, then passing A to `retrieve` will load B, but will not necessarily fill B's fields with their data store values.

## 8.5. Lifecycle Examples

---

### *Example 8.1. Persisting Objects*

```
// create some objects
Magazine mag = new Magazine ("1B78-YU9L", "JavaWorld");

Company pub = new Company ("Weston House");
pub.setRevenue (1750000D);
mag.setPublisher (pub);
pub.addMagazine (mag);

Article art = new Article ("JDO Rules!", "Transparent Object Persistence");
art.setAuthor (new Person ("Fred", "Hoyle"));
mag.addArticle (art);

// we only need to make the root object persistent; JDO will traverse
// the object graph and make all related objects persistent too
PersistenceManager pm = pmFactory.getPersistenceManager ();
pm.currentTransaction ().begin ();
pm.makePersistent (mag);
pm.currentTransaction ().commit ();

// or we could continue using the persistence manager...
pm.close ();
```

### *Example 8.2. Updating Objects*

```
// assume we have an object id for the magazine we want to update
Object oid = ...;

// read a magazine; note that in order to read objects outside of
// transactions you must have the NonTransactionalRead option set
PersistenceManager pm = pmFactory.getPersistenceManager ();
Magazine mag = (Magazine) pm.getObjectById (oid, false);
Company pub = mag.getPublisher ();

// updates should always be made within transactions; note that
// there is no code explicitly linking the magazine or company
// with the transaction; JDO automatically tracks all changes
pm.currentTransaction ().begin ();
mag.setIssue (23);
company.setRevenue (1750000D);
pm.currentTransaction ().commit ();

// or we could continue using the persistence manager...
pm.close ();
```

### *Example 8.3. Deleting Objects*

```
// assume we have an object id for the magazine whose articles
// we want to delete
Object oid = ...;

// read a magazine; note that in order to read objects outside of
// transactions you must have the NonTransactionalRead option set
PersistenceManager pm = pmFactory.getPersistenceManager ();
Magazine mag = (Magazine) pm.getObjectById (oid, false);

// deletes should always be made within transactions
pm.currentTransaction ().begin ();
pm.deletePersistentAll (mag.getArticles ());
pm.currentTransaction ().commit ();

// or we could continue using the persistence manager...
pm.close ();
```

## 8.6. 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.

```
public Class getObjectIdClass (Class pcClass);
```

Returns the JDO identity class used for the given persistence-capable class.

```
public Object newObjectIdInstance (Class pcClass, String identityString);
```

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.

```
public Object getObjectId (Object pc);
```

Returns the JDO identity object for a persistent instance managed by this `PersistenceManager`.

```
public Object getObjectById (Object oid, boolean validate);
```

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.

If the `validate` parameter of this method is set to `true`, then the JDO implementation will throw a `JDODataStoreException` if the data store record for the given JDO identity does not exist. Otherwise, the implementation may return a cached object without validating that it has not been deleted by another persistence manager. Some implementations might return a hollow instance even when no cached object exists, and an exception will not be thrown until you attempt to access one of the object's persistent fields.

### Note

Kodo JDO always throws an exception if `getObjectById` is called for an object that is not in the cache and does not exist in the data store.

## 8.7. Extent Factory

---

```
public Extent getExtent (Class pcClass, boolean includeSubclasses);
```

**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 cannot retrieve Extents for persistence-capable classes whose metadata specifies a value of `false` for the `requires-extent` attribute.

## 8.8. Query Factory

---

```
public Query newQuery ();  
public Query newQuery (Class candidateClass);  
public Query newQuery (Extent candidates);  
public Query newQuery (Class candidateClass, Collection candidates);  
public Query newQuery (Class candidateClass, String filter);  
public Query newQuery (Class candidateClass, Collection candidates, String filter);  
public Query newQuery (Extent candidates, String filter);  
public Query newQuery (String language, Object serialized);  
public Query newQuery (Object serialized);
```

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

## 8.9. Closing

---

```
public boolean isClosed ();  
public void close ();
```

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

---

# Chapter 9. Transaction

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

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

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

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

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

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

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

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

## 9.1. Transaction Types

---

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

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

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

## 9.2. The JDO Transaction Interface

<i><b>Transaction</b></i>
Synchronization Synchronization
boolean NonTransactionalRead
boolean NonTransactionalWrite
boolean RetainValues
boolean RestoreValues
boolean Optimistic
<i>void begin()</i>
<i>void commit()</i>
<i>void rollback()</i>
<i>boolean isActive()</i>

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.

```
public boolean getNontransactionalRead ();
public void setNontransactionalRead (boolean read);
public boolean getNontransactionalWrite ();
public void setNontransactionalWrite (boolean write);
public boolean getRetainValues ();
public void setRetainValues (boolean retain);
public boolean getRestoreValues ();
public void setRestoreValues (boolean restore);
public boolean getOptimistic ();
public void setOptimistic (boolean optimistic);
public Synchronization getSynchronization ();
public void setSynchronization (Synchronization synch);
```

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.

```
public void begin ();
public void commit ();
public void rollback ();
```

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.

```
public boolean isActive ();
```

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 ();
    trans.begin ();
    try
    {
        from.decrementAccount (amnt);
        to.incrementAccount (amnt);
        trans.commit ();
    }
    catch (JDOFatalException jfe) // trans is already rolled back
    {
        throw jfe;
    }
    catch (RuntimeException re) // includes non-fatal JDO exceptions
    {
        trans.rollback (); // or could attempt to fix error and retry
        throw re;
    }
}
```

---

# Chapter 10. Extent

**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.

```
public Class getCandidateClass ();
public boolean hasSubclasses ();
```

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

```
public Iterator iterator ();
public void close (Iterator itr);
public void closeAll ();
```

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`.

## Note

In its default configuration, Kodo JDO automatically uses scrollable JDBC `ResultSet`s when large data sets are being iterated. Combined with Kodo JDO's memory-sensitive data structures, this allows you to efficiently iterate over huge data sets -- even when the entire data set could not possibly fit into memory at once. These scrollable results consume database resources, however, so you are strongly encouraged to close your iterators when you are through with them. If they are not closed immediately, they will be closed when they are garbage collected by the JVM.

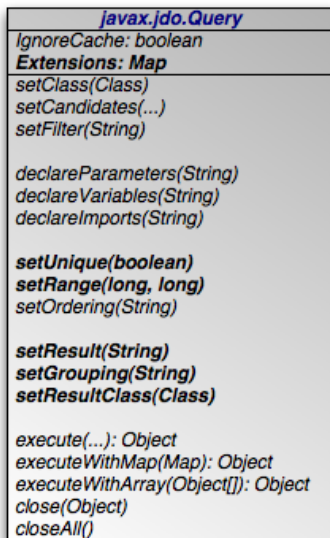
### Example 10.1. Iterating an Extent

```
PersistenceManager pm = ...;

Extent employees = pm.getExtent (Employee.class, true);
Iterator itr = employees.iterator ();
try
{
    while (itr.hasNext ())
        processEmployee ((Employee) itr.next ());
}
finally
{
    employees.close (itr);
}
```

---

# Chapter 11. Query



The `javax.jdo.Query` interface serves two functions: object filtering and data retrieval. This chapter examines each in turn.

## Note

Much of the functionality we discuss in this chapter is new to JDO 2. Though Kodo supports all of the features defined in the following sections, many JDO implementations may not. Additionally, because official JDO 2 jars are not yet available, you will have to cast your query objects to `kodo.query.KodoQuery` to access any JDO 2 APIs. The UML diagram above depicts these APIs in bold. For simplicity, casts have been left out of the example code throughout the chapter.

We describe all JDO 2 features as they appear in the JDO 2 Early Draft specification and subsequent JCP Expert Group proposals. Some of these features may change before the JDO 2 specification is finalized.

## 11.1. Object Filtering

JDO queries evaluate a group of candidate objects against a set of conditions, eliminating the objects that don't match. We refer to this as *object filtering*. The original group of objects might be a `Collection` of instances you've already retrieved, or an `Extent`. Recall from [Chapter 10, Extent](#) [53] that an `Extent` represents all persistent instances of a certain class, optionally including subclasses.

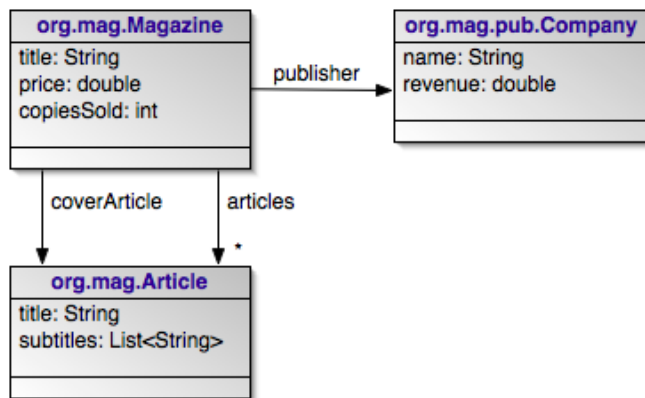
Conditions are specified in the JDO Query Language (JDOQL). The filtering process might take place in the datastore, or might be executed in memory. JDO does not mandate any one mechanism, and many implementations use a mixture of datastore and in-memory execution depending on the circumstances.

Basic object filtering utilizes the following `Query` methods:

```
public void setClass (Class candidateClass);
public void setCandidates (Extent candidates);
public void setCandidates (Collection candidates);
public void setFilter (String filter);
```

- `setClass` names the candidate class. Candidate objects that are not assignable to the candidate class cannot match the filter.
- The two `setCandidates` methods specify the set of objects to evaluate. If you supply an `Extent`, you don't need to also call `setClass`; the query inherits the `Extent`'s candidate class. Reciprocally, if you call `setClass` but do not call either `setCandidates` method, the query candidates default to the `Extent` of the candidate class, including subclasses.
- `setFilter` accepts a JDOQL string establishing the conditions an object must meet to be included in the query results. As you'll see shortly, JDOQL looks exactly like a Java boolean expression using the candidate class' persistent fields and relations. When you don't set a filter explicitly, it defaults to the simplest possible boolean expression: `true`. In this case, all candidate objects assignable to the candidate class match the query.

Let's see an example of basic object filtering in action. Our example draws on the following object model, which we continue to use throughout the chapter.



### Example 11.1. Filtering

The following code processes all `Magazine` objects in the database whose price is greater than 10 dollars.

```

PersistenceManager pm = ...;
Query query = pm.newQuery ();
query.setClass (Magazine.class);
query.setCandidates (pm.getExtent (Magazine.class, true));
query.setFilter ("this.price > 10");
Collection mags = (Collection) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
  
```

The code above is technically correct, but could be much more concise. First, remember that setting a candidate class defaults the candidates to the extent of that class, and vice versa. So we don't need both the `setClass` and `setCandidates` calls. We can also get rid of the `this` qualifier on `price` in our filter string, since unqualified field names are already assumed to belong to the current object, just as in Java code. Finally, we can take advantage of the fact that `PersistenceManager` overloads the `newQuery` method to tighten our code further. Here is the revised version:

```

PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "price > 10");
Collection mags = (Collection) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
  
```

The filter above is rather basic. Before moving on to more advanced queries, though, we need to dive into the details of JDOQL.

## 11.2. JDOQL

---

JDOQL is a datastore-neutral query language based on Java. It relies solely on your object model, completely abstracting the details of the underlying data model. As you have already seen, the syntax of JDOQL is virtually identical to Java syntax. In fact, a good trick when writing a JDOQL string is to pretend that you are writing a method of the query's candidate class. You have access to all of the class' persistent fields, and to the `this` keyword. There are, however, some notable ways in which JDOQL differs from Java:

- You can traverse through private persistent fields of relations.
- String literals can be either double-quoted or single-quoted.
- Equality and ordering comparisons between primitives and instances of wrapper classes (`Boolean`, `Byte`, `Integer`, etc) are valid.
- Equality and ordering between `Dates` are valid.
- Equality comparisons always use the `==` operator; the `equals` method is not supported.
- The assignment operators (`=`, `+=`, `*=`, etc) as well as the `++` and `--` operators are not supported.
- Methods are not supported, with the following exceptions:
  - `Collection.contains`
  - `Collection.isEmpty`
  - **`Map.containsKey`**
  - **`Map.containsValue`**
  - **`Map.isEmpty`**
  - **`String.toUpperCase`**
  - **`String.toLowerCase`**
  - `String.startsWith`
  - `String.endsWith`
  - **`String.matches`**: Regular expression support is limited to the use of `.`, `*`, and `(?i)` to represent case-insensitive matching.
  - **`String.indexOf`**: Both the `String.indexOf(String str)` and the `String.indexOf(String str, int fromIndex)` forms are supported.
  - **`String.substring`** : Both the `String.substring(int start)` and the `String.substring(int beingIndex, int endIndex)` forms are supported.
  - **`Math.abs`**
  - **`Math.sqrt`**
  - **`JDOHelper.getObjectId`**

- JDOQL supports the following aggregates: **min**, **max**, **sum**, **avg**, **count**. We discuss aggregates in detail later in the chapter.
- Traversing a null-valued field causes the subexpression to evaluate to false rather than throwing a `NullPointerException`.

### Note

Bold items represent JDO 2's additions to JDOQL. Kodo also offers several proprietary extensions to JDOQL, and allows you to define your own extensions. See the Reference Guide [Section 10.7, “Query Extensions”](#) [290] and [Section 13.3, “JDOQL Subqueries”](#) [323] for details.

While it is important to note the differences between JDOQL and Java enumerated in the list above, it is just as important to note what is *not* in the list. Mathematical operators, logical operators, `instanceof`, casting, field traversal, and static field access are omitted, meaning they are all fully supported by JDOQL.

We demonstrate some of the interesting aspects of JDOQL below.

### Example 11.2. Relation Traversal and Mathematical Operations

Find all magazines whose sales account for over 1% of the total revenue for the publisher. Notice the use of standard Java "dot" notation to traverse into the persistent fields of `Magazine`'s `publisher` relation.

```
Query query = pm.newQuery (Magazine.class,
    "price * copiesSold > publisher.revenue * .01");
Collection mags = (Collection) query.execute ();
```

### Example 11.3. Precedence, Logical Operators, and String Functions

Find all magazines whose publisher's name is "Random House" or matches a regular expression, and whose price is less than or equal to 10 dollars. Here, we use single-quoted string literals to avoid having to escape double quotes within the filter string. Notice also that we compare strings with `==` rather than the `equals` method.

```
Query query = pm.newQuery (Magazine.class, "price <= 10.0 "
    + "&& (publisher.name == 'Random House' "
    + "|| publisher.name.toLowerCase ().matches ('add.*ley'))");
Collection mags = (Collection) query.execute ();
```

### Example 11.4. Collections

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

```
Query query = pm.newQuery (Magazine.class,
    "coverArticle.subtitles.contains ('The Real Story') "
    + "|| coverArticle.subtitles.isEmpty ()");
```

```
Collection mags = (Collection) query.execute ();
```

### ***Example 11.5. Static Methods***

Retrieve all magazines whose publisher's revenue's square root is greater than 100 dollars.

```
Query query = pm.newQuery (Magazine.class,  
    "Math.sqrt (publisher.revenue) > 100");  
Collection mags = (Collection) query.execute ();
```

## **11.3. Advanced Object Filtering**

---

In this section, we explore advanced topics in object filtering, supported by the following Query methods:

```
public void declareParameters (String parameters);  
public void declareVariables (String variables);  
public void declareImports (String imports);
```

- The `declareParameters` method defines the names and types of the query's parameters. Declaring query parameters is analogous to declaring the parameter signature of a Java method, and uses the same syntax. Parameters acts as placeholders in the filter string, allowing you to supply new values on each execution. Parameters also allow for "query by example", as demonstrated in **Example 11.8, "Query By Example" [60]**

Parameters do not have to be declared. As you will see in the examples below, you can introduce *implicit* parameters into your queries simply by prefixing the parameter name with a colon in your JDOQL string. Wherever the type of an implicit parameter can't be inferred from the context, you can set the type by casting the parameter in the JDOQL.

You cannot mix implicit and declared parameters in the same query. Each query must declare all of its parameters, or none of them.

- `declareVariables` names the local variables used in the query filter. It uses standard Java variable declaration syntax. Query variables are typically used to place conditions on elements of collections or maps.

Like parameters, variables do not have to be declared. Whenever you use an unrecognized identifier where a variable would be appropriate, the JDO implementation will dynamically define an *implicit* variable with that name. In the case of an unbound variable, you must cast the variable in your JDOQL to supply its type. Unlike parameters, you can mix both declared and implicit variables in the same query.

- You can import classes into the query's namespace with the `declareImports` method. The method argument uses standard Java `import` syntax. By default, queries only recognize unqualified class names when the class is in the package of the candidate class, or when it is in `java.lang`. Using imports, you can save yourself the trouble of typing out full class names in your parameter and variable declarations, and in your your filter string (class names can appear in filter strings when you use the `instanceof` operator, access a static field, or perform a cast).

## Note

Implicit parameters and variables are a new feature of JDO 2. JDO 1 implementations require you to declare all parameters and variables.

The following examples will give you a feel for the query elements described above. They use the object model we defined in the [previous section](#).

### *Example 11.6. Imports and Declared Parameters*

Find the magazines with a certain publisher and title, where the publisher and title are supplied as parameters on each execution.

```
PersistenceManager pm = ...;
Company comp = ...;
String str = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher == pub && title == ttl");
query.declareImports ("import org.mag.pub.*");
query.declareParameters ("String ttl, Company pub");
Collection mags = (Collection) query.execute (str, comp);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

There are two things to take away from this example. First, the import prevents us from having to qualify the `Company` class name when declaring our `pub` parameter (although in this case, it would have been easier to just write out the full class name in the parameter declaration). The unqualified `Company` name is not automatically recognized by the query because `Company` isn't in the same package as `Magazine`, the candidate class.

Second, notice that we supply values for the parameter placeholders when we execute the query. Parameters can be of any type recognized by JDO, though primitive parameters have to be supplied as instances of the appropriate wrapper type on execution. The `Query` interface includes several methods for executing queries with various numbers of parameters. When you use declared parameters, you should supply the parameter values in the same order that you declare the parameters.

### *Example 11.7. Implicit Parameters*

The query below is exactly the same as our previous example, except this time we use implicit parameters.

```
PersistenceManager pm = ...;
Company comp = ...;
String str = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher == :pub && title == :ttl");
Collection mags = (Collection) query.execute (comp, str);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

Here, we use colon-prefixed names to introduce new parameters without declarations. When we execute the query, we supply the parameter values in the order the implicit parameters first appear in the JDOQL. Later in the chapter, we'll see queries that consist of multiple JDOQL strings. Each string might introduce new implicit parameters. When deciding the proper order to supply the

parameter values, start with the parameters in the result string, then those in the filter, then the grouping string, and finally the ordering string.

In the query above, we can infer the type of each parameter based on its context. This is almost always the case. There are times, however, when the context alone is not enough to determine the type of an implicit parameter. In these cases, use a cast to supply the parameter type:

```
PersistenceManager pm = ...;
Company comp = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher.revenue == ((org.mag.pub.Company) :pub).revenue");
Collection mags = (Collection) query.execute (comp);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

### ***Example 11.8. Query By Example***

Parameters do not need to be in the datastore to be useful. You can implement "query by example" in JDO by using an existing "example" object as a query parameter:

```
PersistenceManager pm = ...;
Magazine example = new Magazine ();
example.setPrice (100);
example.setTitle ("Fourier Transforms");
Query query = pm.newQuery (Magazine.class,
    "price == ex.price && title == ex.title");
query.declareParameters ("Magazine ex");
Collection mags = (Collection) query.execute (example);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
```

### ***Example 11.9. Variables***

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

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "articles.contains (art) "
    + "&& art.title == 'Fourier Transforms'");
query.declareVariables ("Article art");
Collection mags = (Collection) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

A variable represents any persistent instance of its declared type. So you can read the filter string above as: "The magazine's articles collection contains some article art, where art's title is 'Fourier Transforms'". Notice how we bind art to a particular collection with the `contains` method, then test its properties in an `&&`'d expression. This is a common pattern in JDOQL filters, and applies equally well to placing conditions on the keys and values of maps.

Of course, we don't have to declare art explicitly. The same query without `declareVariables` would work just as well:

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "articles.contains (art) "
    + "&& art.title == 'Fourier Transforms'");
Collection mags = (Collection) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

In this case the JDO implementation assumes `art` is an implicit variable because it does not match the name of any field in the candidate class. The new variable's type is set to the known element type of the collection that contains it.

### Example 11.10. Unbound Variables

The example above uses a variable to represent any element in a collection. We refer to variables used to test collection or map elements as *constrained* or *bound* variables, because the values of the variable are limited by the collection or map involved. Many JDO implementations also support *unbound* variables. Rather than representing a collection or map element, an unbound variable represents any persistent instance of its class. Consider the following example:

```
Query query = pm.newQuery (Article.class, "mag.copiesSold > 10000 "
    + "&& mag.coverArticle == this");
query.declareVariables ("Magazine mag");
Collection arts = (Collection) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

What does this query do? Let's break it down. The first clause matches any magazine that has sold more than 10,000 copies. The second clause requires that the cover article of the magazine is the candidate instance being evaluated (notice the query's candidate class is `Article` in this example). So the query returns all articles that are the cover article for a magazine that has sold more than 10,000 copies. The unbound variable `mag` allowed us to overcome the fact that there was no direct relation from `Article` to `Magazine` (only the reverse).

Later in this chapter, we'll see how to use a *projection* to drastically simplify this query.

We can also execute this query without declaring the `mag` variable explicitly. Without a constraining `contains` clause, however, the type of an unbound, implicit variable is impossible to infer. Use a cast to supply the type:

```
Query query = pm.newQuery (Article.class, "((Magazine) mag).copiesSold > 10000 "
    + "&& mag.coverArticle == this");
Collection arts = (Collection) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

Up until now, we have focused on how to configure our queries with the right filter, but we have ignored how to actually execute the query. The next section corrects this oversight.

## 11.4. Compiling and Executing Queries

```
public void compile ();
```

Compiling is a hint to the implementation to optimize an execution plan for the query. During compilation, the query validates all of its elements and reports any inconsistencies by throwing a `JDOUserException`. Queries automatically compile themselves before they execute, so you probably won't use this method very often.

```
public Object execute ();
public Object execute (Object param1);
public Object execute (Object param1, Object param2);
public Object execute (Object param1, Object param2, Object param3);
public Object executeWithArray (Object[] params);
public Object executeWithMap (Map params);
```

As evident from the examples throughout this chapter, queries are executed with one of the many `execute` methods defined in the `Query` interface. If your query uses between 0 and 3 parameters, you can use the `execute` version that takes the proper number of `Object` arguments. For queries with more than 3 parameters, you can use the generic `executeWithArray` and `executeWithMap` methods (in the latter method, the map is keyed on the parameter name).

All `execute` methods declare a return type of `Object` because the return type depends on how you've configured the query. So far we've only seen queries that return `Collections`, but this will change in the next section.

```
public void close (Object result);
public void closeAll ();
```

Some query results may hold on to datastore resources. Therefore, you should close query results when you no longer need them. You can close an individual query result with the `close` method, or all open results at once with `closeAll`.

### Note

Collection query results from Kodo JDO always implement the `java.util.List` interface. This allows you to access the results in any order. By default, Kodo greedily fills each result `List` and immediately releases all database resources, making `close` unnecessary.

You can, however, configure Kodo to use lazy element instantiation where appropriate. Under lazy instantiation, list elements that are never accessed are never created, and elements may be released after being traversed. Combined with Kodo's memory-sensitive data structures and smart eager fetching policies, lazy element instantiation also allows you to efficiently iterate over huge data sets -- even when the entire data set could not possibly fit into memory at once. These lazy results consume database resources, though, so if you enable them you are strongly encouraged to close all query results after processing. If they are not closed explicitly, they are closed automatically when the JVM garbage collects them. See [Section 4.10, "Large Result Sets" \[180\]](#) in the Reference Guide for details on configuring Kodo for lazy result instantiation.

## 11.5. Limits and Ordering

We have seen how you tell the query *which* objects you want. This section shows you how to also tell it *how many* objects you want, and *what order* you want them in.

```
public void setUnique (boolean unique);
public void setRange (long start, long end);
```

Use the `setUnique` method when you know your query matches at most a single object. A unique query always returns either

the one candidate instance that matched the filter, or null; it never returns a `Collection`. In fact, unique queries enforce the one-result rule by throwing a `JDOUserException` if more than one candidate survives the filtering process.

While `setUnique` is designed for queries with only one result, `setRange` is designed for those with many. Most applications that deal with large amounts of data only process or display a fixed number of records at a time. For example, a web app might show the user 20 results per page, though many thousands of records exist in the database. Using `setRange`, you can retrieve the exact range of objects you're interested in. The method behaves much like `List.subList` or `String.substring`: it uses 0-based indexes, and the end index is exclusive. Unlike these methods, however, `setRange` won't throw an exception when your range extends beyond the available elements. If 50 records exist and you ask for the range 40, 60, you'll receive a collection containing the 41st (remember indexes are 0-based) through 50th result objects. If you ask for the range 60, 80, you'll get an empty collection.

Whenever you don't specify a range on a query, it defaults to a start index of 0 and an end index of `Long.MAX_VALUE`, which represents no limit.

```
public void setOrdering (String ordering);
```

Ranges are meaningless if the result order isn't constant between query executions, and many datastores don't guarantee consistent natural ordering. `setOrdering` gives you control over the order of results, so you don't have to rely on the vagaries of the datastore. The method's argument is a comma-separated list of field names or expressions, each followed by the keyword `ascending` or `descending`. Those keywords can be abbreviated as `asc` and `desc` respectively. Results are ordered primarily by the first (left-most) expression. Wherever two results compare equal with that expression, the next ordering expression is used to order them, and so on.

### ***Example 11.11. Unique***

Find the magazine "Spaces" published by Manning. We know that this query can only match 0 or 1 magazines, so we set the `unique` flag to avoid having to extract the result object from a `Collection`.

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class,
    "publisher.name == 'Manning' && title == 'Spaces'");
query.setUnique (true);
processMagazine ((Magazine) query.execute ());
```

### ***Example 11.12. Result Range and Ordering***

Order all magazines by price and return the 21st through 40th results.

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class);
query.setOrdering ("price ascending");
query.setRange (20, 40);
Collection mags = (Collection) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

## 11.6. Projections

At the beginning of the chapter, we said that JDO queries serve two purposes: object filtering and data retrieval. So far, we have concentrated primarily on object filtering. We now shift our focus to data retrieval, starting with projections.

A *projection* consists of one or more values formed by traversing the fields of the candidates. You can use projections to get back only the fields or relations that you are interested in, rather than manually pulling the desired data from each result object. As you will see later, you can even instruct JDO to pack the projected field values into a class of your choosing.

Projections are established with the `setResult` method:

```
public void setResult (String result);
```

`result` is a comma-separated string of result expressions. Each result expression might be a field name, a relation traversal, or any other valid JDOQL clause. In all our queries to this point, we've taken advantage of the fact that when you don't specify a result string explicitly, it defaults to `this`, meaning the query returns each matching object itself (actually, `result` defaults to `distinct this as C`, where `C` is the unqualified candidate class name, but for now just think of it as `this`).

We present a simple projection below. We're still using the object model defined in [Section 11.1, “Object Filtering”](#) [54].

### *Example 11.13. Projection*

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("title, price");
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) itr.next ();
    processData ((String) data[0], (Double) data[1]);
}
query.close (results);
```

Each query result in this case is not a `Magazine` instance, but an array consisting of each magazine's title and price. Projections allow you to acquire only the data you're interested in, and ignore the rest.

`JDOHelper.getObjectId` becomes a particularly useful JDOQL method when applied to projections. Notice that because we only select a single result expression this time, we get back the raw result values (in this case, identity objects) instead of object arrays.

```
Query query = pm.newQuery (Magazine.class, "publisher.revenue > 1000000");
query.setResult ("JDOHelper.getObjectId (this)");
Collection oids = (Collection) query.execute ();
for (Iterator itr = oids.iterator (); itr.hasNext ();)
    processObjectId (itr.next ());
query.close (oids);
```

Projections aren't limited to the candidate class and its fields. Here is a query selecting the first 3 characters of the cover article's title for all magazines under 5 dollars:

### *Example 11.14. Projection Field Traversal*

```
Query query = pm.newQuery (Magazine.class, "price < 5");
query.setResult ("coverArticle.title.substring (0, 3)");
Collection ttls = (Collection) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processTitle ((String) itr.next ());
query.close (ttls);
```

But what if we change our minds, and decide we're really interested in the *subtitles* of all cover articles, rather than their titles? Setting the result query to `coverArticle.subtitles.substring (0, 3)` won't do the trick; `subtitles` is a `Collection` field, and `Collection` doesn't have a `substring` method. What we're after isn't the `subtitles` field itself, but its elements.

You may recall from previous examples that we used query variables to represent the elements of collections. We can apply the same solution here. By binding a variable to the elements of the `subtitles` collection in our filter and selecting the variable in our result string, we achieve our goal.

### Example 11.15. Projection Variables

```
Query query = pm.newQuery (Magazine.class, "price < 5 "
    + "&& coverArticle.subtitles.contains (ttl)");
query.setResult ("ttl.substring (0, 3)");
Collection ttls = (Collection) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processSubtitle ((String) itr.next ());
query.close (ttls);
```

Now we have our subtitle data, but we're faced with a new problem: a lot of subtitles start with the same 3 characters, and we really only want to process each string once. Luckily, JDO has a built-in solution. If you begin a result string with the `distinct` keyword, JDO filters out duplicates. This applies not only to simple projections as in this example, but to complex projections consisting of multiple return values, including persistent object values.

### Example 11.16. Distinct Projection

```
Query query = pm.newQuery (Magazine.class, "price < 5 "
    + "&& coverArticle.subtitles.contains (ttl)");
query.setResult ("distinct ttl.substring (0, 3)");
Collection ttls = (Collection) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processSubtitle ((String) itr.next ());
query.close (ttls);
```

## Note

Some object-relational mapping products require you to alter your query to circumvent duplicate results caused by relational joins. In the simple case, this may only involve using a `SELECT DISTINCT` equivalent. But in the complex case, such as with aggregate data, you might need to tell the implementation to use subselects and other complicated workarounds to avoid the relational joins problem.

This is not the case in JDO. JDO implementations always automatically eliminate duplicates caused by relational joins. You only need to use the `distinct` keyword in your JDO result string when there are repeated values in the database that you'd like to filter out.

Remember the unbound variable example earlier in this chapter? We wanted to find all the cover articles of magazines that sold over 10,000 copies, but we had to work around the fact that there is no relation from `Article` to `Magazine`:

```
Query query = pm.newQuery (Article.class, "mag.copiesSold > 10000 "
    + "&& mag.coverArticle == this");
query.declareVariables ("Magazine mag");
Collection arts = (Collection) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

To conclude the section, let's simplify this query using our newfound knowledge of projections:

```
Query query = pm.newQuery (Magazine.class, "copiesSold > 10000");
query.setResult ("coverArticle");
Collection arts = (Collection) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

## 11.7. Aggregates

Aggregates are just what they sound like: aggregations of data from multiple instances. Combined with object filtering and grouping, aggregates are powerful tools for summarizing your persistent data.

JDOQL includes the following aggregate functions:

- `min(expression)`: Returns the minimum value of the given expression among matching instances.
- `max(expression)`: Returns the maximum value of the given expression among matching instances.
- `sum(expression)`: Returns the sum of the given expression over all matching instances.
- `avg(expression)`: Returns the average of the given expression over all matching instances.
- `count(expression)`: Returns the number of matching instances for which the given expression is not null.

### Note

Kodo allows you to define your own aggregate functions. See [Section 10.7, “Query Extensions” \[290\]](#) in the Reference Guide for details.

The following example counts the number of magazines that cost under 5 dollars:

#### *Example 11.17. Count*

```
Query query = pm.newQuery (Magazine.class, "price < 5");
query.setResult ("count(this)");
Long count = (Long) query.execute ();
```

You may be thinking that we could have gotten the count just as easily by executing a standard query and calling `Collection.size ()` on the result, and you'd be right. But not all aggregates are so easy to replace. Our next example retrieves the minimum, maximum, and average magazine prices in the database. These values would be a little more difficult to calculate manually. More importantly, iterating over every single persistent magazine in order to factor its price into our calculations would be woefully inefficient.

### ***Example 11.18. Min, Max, Avg***

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("min(price), max(price), avg(price)");
Object[] prices = (Object[]) query.execute ();
Double min = (Double) prices[0];
Double max = (Double) prices[1];
Double avg = (Double) prices[2];
```

The functionality described above is useful, but aggregates only really shine when you combine them with object grouping.

```
public void setGrouping (String grouping);
```

The `Query` interface's `setGrouping` method allows you to group query results on field values. The grouping string consists of one or more comma-separated clauses to group on, optionally followed by the `having` keyword and a boolean expression. The `having` expression pares down the candidate groups just as the query's filter pares down the candidate objects.

Now your aggregates apply to *each group*, rather than to all matching objects. Let's see this in action:

### ***Example 11.19. Grouping***

The following query returns each publisher and the average price of its magazines, for all publishers that publish no more than 10 magazines.

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("publisher, avg(price)");
query.setGrouping ("publisher having count(this) <= 10");
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) itr.next ();
    processData ((Company) data[0], (Double) data[1]);
}
query.close (results);
```

You probably noticed that in our initial aggregate examples, the queries all returned a single result object, while the query above returned a `Collection`. Before you get too confused, let's take a brief detour to examine query return types.

- If you have called `setUnique(true)`, the query returns a single result object (or null).
- Else if you have called `setUnique(false)`, the query returns a collection.
- Else if the query result is an aggregate and you have not specified any grouping, the query returns a single result object.
- Else the query returns a collection.

In addition to how many results are returned, query configuration can affect the type of each result:

- If you have specified a result class, the query returns instances of that class. We cover result classes in the next section.
- Else if you have not set a result string, the query returns instances of the candidate class.
- Else if you have specified a single projection or aggregate result:
  - A projection returns instances of the projected field type. When mathematical expressions are involved, the less precise operand is always promoted to the type of the more precise operand.
  - `min(expression)` returns an instance of the expression type.
  - `max(expression)` returns an instance of the expression type.
  - `sum(expression)` returns a `long` for integral types other than `BigInteger`, and the expression type for all other types.
  - `avg(expression)` returns an instance of the expression type.
  - `count(expression)` returns a `long`.
- Else if you have specified multiple projections or aggregates in your result string, the query returns instances of `Object[]`, where the class of each array index value follows the typing rules above.
- Casting a projection or aggregate expression in the result string converts that result element to the type specified in the cast.

Don't worry about trying to memorize all of these rules. In practice, they amount to a much simpler rule: queries return what you expect them to. A query for all the magazines that match a filter returns a collection of `Magazines`. But an aggregate query for the count of all magazines that match a filter just returns a `Long`. A projection query for the title of all magazines returns a collection of `Strings`. But a projection for both the title and price of each magazine returns a collection of `Object[]`s, each consisting of a `String` and a `Double`. So although you can always explicitly set the `unique` flag and result class to obtain a specific result shape, the defaults are usually exactly what you want.

## 11.8. Result Class

---

Queries have the ability to pack result data into a custom result class. You might use this feature for anything from populating data transfer objects automatically, to avoiding the casting and other inconveniences involved in dealing with the `Object[]`s normally generated by multi-valued projections. You specify a custom result class with the `setResultClass` method.

```
public void setResultClass (Class resultClass);
```

## 11.8.1. JavaBean Result Class

JDO populates result objects by matching the result class' public fields and JavaBean setter methods to the expressions defined in the query result string. The result class must be public (or otherwise instantiable via reflection), and must have a no-args constructor.

### *Example 11.20. Populating a JavaBean*

```
public class SalesData
{
    private double price;
    private int copiesSold;

    public void setPrice (double price)
    {
        this.price = price;
    }

    public double getPrice ()
    {
        return price;
    }

    public void setCopiesSold (int copiesSold)
    {
        this.copiesSold = copiesSold;
    }

    public int getCopiesSold ()
    {
        return copiesSold;
    }
}

Query query = pm.newQuery (Magazine.class);
query.setResult ("price, copiesSold");
query.setResultClass (SalesData.class);
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processSalesData ((SalesData) itr.next ());
query.close (results);
```

The example above is simple enough; the names of the projected fields are matched to the setter methods in the result class. But what if the names don't match? What if the result expressions contain aggregates, mathematical expressions, and relation traversals, all of which contain symbols that *can't* match a field or setter method name?

JDO provides the answer in the form of result expression *aliases*. An alias is a label assigned to a particular result expression for the purposes of matching that expression to fields or methods in the result class. To demonstrate this, let's modify our example above to populate each `SalesData` object not with the price and copies sold of each magazine, but with the average price and total copies sold of all magazines published by each company.

### *Example 11.21. Result Aliases*

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("avg(price) as price, sum(copiesSold) as copiesSold");
query.setResultClass (SalesData.class);
query.setGrouping ("publisher");
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processSalesData ((SalesData) itr.next ());
query.close (results);
```

Earlier in this chapter, we mentioned that the default result string for a query is `distinct this as C`, where `C` is the unqualified name of the candidate class. Now, finally, the meaning of this default string should be clear. But just to make it concrete, here is an example:

### *Example 11.22. Taking Advantage of the Default Result String*

```
public class Wrapper
{
    private Magazine mag;

    public void setMagazine (Magazine mag)
    {
        this.mag = mag;
    }

    public Magazine getMagazine ()
    {
        return mag;
    }
}

Query query = pm.newQuery (Magazine.class);
query.setResultClass (Wrapper.class);
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processWrapper ((Wrapper) itr.next ());
query.close (results);
```

In this query on the `org.mag.Magazine` class, the default result string is `distinct this as Magazine`. Because our result class has a corresponding `setMagazine` method, the query can automatically populate each `Wrapper` with a matching magazine.

## 11.8.2. Generic Result Class

Whenever the specified result class does not contain a public field or setter method matching a particular result expression, the query looks for a method named `put` that takes two `Object` arguments. If found, the query invokes that method with the result expression or alias as the first argument, and its value as the second argument. This not only means that you can include a generic `put` method in your custom result classes, but that any `Map` implementation is suitable for use as a query result class.

### *Example 11.23. Populating a Map*

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("title.toUpperCase (), copiesSold");
query.setResultClass (HashMap.class);
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    HashMap map = (HashMap) itr.next ();
    processData (map.get ("title.toUpperCase ()"), map.get ("copiesSold"));
}
query.close (results);
```

## 11.9. Single-String JDOQL

It is often convenient to represent an entire JDOQL query as a single string. JDO defines a single-string query form that encom-

passes all parts of a query: the unique flag, result string, result class, filter, parameters, variables, imports, grouping, ordering, and range.

## Note

Single-string JDOQL is new in JDO 2. Its format is subject to change prior to the final JDO 2 specification. In addition to standard single-string JDOQL support, Kodo uses the single-string format to implement JDOQL subquery support, covered in [Section 13.3, “JDOQL Subqueries” \[323\]](#)

We present the grammar for single-string JDOQL queries below. Clauses in square brackets are optional. Clauses in angle brackets are required, and represent portions of the Query API.

```
select [unique] [<result>] [into <result class>]
  [from <candidate class> [exclude subclasses]]
  [where <filter>]
  [parameters <parameter declarations>]
  [variables <variable declarations>]
  [imports <import declarations>]
  [group by <grouping>]
  [order by <ordering>]
  [range <start>, <end>]
```

The `PersistenceManager` has a factory method designed specifically for constructing a query from a single argument (in this case, a JDOQL string):

```
public Query newQuery (Object query);
```

In each code block below, we create and execute a standard JDOQL query, then demonstrate the same process using the single-string format. As you'll see, translating any query into single-string form is just a matter of plugging the corresponding strings into the grammar above.

```
Query query = pm.newQuery ();
query.setResult ("distinct title");
query.setClass (Magazine.class);
query.setFilter ("price < :p");
query.setRange (10, 20);
Collection results = (Collection) query.execute (new Double (10.0));

Query query = pm.newQuery ("select distinct title from org.mag.Magazine "
+ "where price < :p range 10, 20");
Collection results = (Collection) query.execute (new Double (10.0));
```

```
Query query = pm.newQuery ();
query.setUnique (true);
query.setCandidates (pm.getExtent (Magazine.class, false));
query.setFilter ("title == 'JDJ'");
Magazine mag = (Magazine) query.execute ();

Query query = pm.newQuery ("select unique from org.mag.Magazine "
+ "exclude subclasses where title == 'JDJ'");
Magazine mag = (Magazine) query.execute ();
```

```
Query query = pm.newQuery ();
query.setResult ("publisher.name as pub, avg(price) as price");
```

```

query.setResultClass (org.mag.pub.PubPrice.class);
query.setClass (Magazine.class);
query.setGrouping ("publisher having avg(price) < :p");
query.setOrdering ("publisher.name ascending");
Collection results = (Collection) query.execute (new Double (10.0));

Query query = pm.newQuery ("select publisher.name as pub, avg(price) as price "
    + "into org.mag.pub.PubPrice from org.mag.Magazine "
    + "group by publisher having avg(price) < :p "
    + "order by publisher.name ascending");
Collection results = (Collection) query.execute (new Double (10.0));

```

On a final note, single-string queries are mutable. You can construct a single-string query as in the examples above, then change the result string, filter, or any other part of the query through the standard Query APIs. This is especially useful for temporary properties like the result range.

## 11.10. Named Queries

Named queries provide a means to define complex or commonly used queries in metadata files. These queries have all the capabilities of queries created in code, including support for parameters, aggregates and projections.

You declare named queries in `.jdoquery` files. The query file format is quite similar to that of **JDO metadata**. In addition, query metadata files are stored in the same **locations** as JDO metadata. The `.jdoquery` XML structure includes attributes and elements which correspond to methods in the **javax.jdo.Query** interface.

```
public Query newNamedQuery (Class cls, String name);
```

At runtime, you obtain named queries with the `PersistenceManager.newNamedQuery` method. The `Class` argument names the query's candidate class. This argument may be null if the query does not have a candidate class, as we'll see below. The `String` argument is the name of the query.

### Note

Until official JDO 2 jars are released, you will have to cast your persistence managers to the `kodo.runtime.KodoPersistenceManager` interface to access the `newNamedQuery` method.

Before returning a named query, the system populates it with the information provided in the `.jdoquery` metadata, then compiles it to make sure it is valid. You can still change the query before executing it, using the methods of the `Query` interface. These changes won't affect other query instances returned from subsequent calls to `newNamedQuery`.

### 11.10.1. Named Query DTD

The Document Type Definition (DTD) for named query metadata is very similar to JDO metadata. We discuss the various elements of the DTD below.

```

<!ELEMENT jdoquery ((query)*, (package)*, (extension)*)>
<!ELEMENT package ((class)+)>
<!ATTLIST package name CDATA #REQUIRED>

<!ELEMENT query ((declare)?, (filter|sql|jdoql)?, (result)?, (extension)*)>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query ignore-cache (true|false) #IMPLIED>
<!ATTLIST query include-subclasses (true|false) #IMPLIED>
<!ATTLIST query filter CDATA #IMPLIED>
<!ATTLIST query sql CDATA #IMPLIED>
<!ATTLIST query jdoql CDATA #IMPLIED>

```

```

<!ATTLIST query ordering CDATA #IMPLIED>
<!ATTLIST query range CDATA #IMPLIED>

<!ELEMENT filter (#PCDATA)>
<!ELEMENT sql (#PCDATA)>
<!ELEMENT jdoql (#PCDATA)>

<!ELEMENT declare (extension)*>
<!ATTLIST declare imports CDATA #IMPLIED>
<!ATTLIST declare parameters CDATA #IMPLIED>
<!ATTLIST declare variables CDATA #IMPLIED>

<!ELEMENT result (#PCDATA)>
<!ATTLIST result unique (true|false) #IMPLIED>
<!ATTLIST result class CDATA #IMPLIED>
<!ATTLIST result grouping CDATA #IMPLIED>

<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>

```

You can define queries just below the root `jdoquery` element or at the `class` level. Queries within the `jdoquery` root do not have an automatic candidate class. These are usually **single-string JDOQL** queries with an inline candidate class declaration, or SQL projections, which we cover in the next chapter. Queries denoted in a `class` element are given that candidate class.

The `query` element details options like the query name, ordering, range, and whether to include subclasses in the results. You can include the filter string as an attribute at this level or as the text of a nested `filter` element.

Use the `declare` element to control variable, parameter, and import declarations.

The `result` element allows you to use aggregates, projections, custom result classes, unique results, and grouping.

Finally, you can forgo most of the elements and attributes above and express the query in single-string form. Use the `jdoql` attribute on the `query` element, or the nested `jdoql` element.

## 11.10.2. Named Query Examples

We will now examine some named query examples. We begin by looking at the metadata for a very simple query.

### *Example 11.24. Defining a Named Query*

The XML below defines a query named `sports` for the `org.mag.Magazine` class. You might place this XML in the `org/mag/package.jdoquery` or `org/mag/Magazine.jdoquery` files.

```

<?xml version="1.0"?>
<jdoquery>
  <package name="org.mag">
    <class name="Magazine">
      <query name="sports" filter="title.startsWith ('Sports')"/>
    </class>
  </package>
</jdoquery>

```

Declaring the named query above is equivalent to the following Java code:

```

PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class);
query.setFilter ("title.startsWith ('Sports')");
query.compile ();

```

Here is a more complex case:

### ***Example 11.25. Ordering, Range, Variables, and Parameters***

```
<xml version="1.0"?>
<jdoquery>
  <package name="org.mag">
    <class name="Magazine">
      <query name="findBySub" ordering="title ascending" range="0,10">
        <declare parameters="String sub" variables="Article art"/>
        <filter>
          articles.contains (art) &amp;&amp; art.subtitles.contains (sub)
        </filter>
      </query>
    </class>
  </package>
</jdoquery>
```

This query finds the first 10 magazines that have an article with a subtitle equal to `sub`, where `sub` is a string whose value you supply on execution. The results are ordered alphabetically on the magazine title. Note that we declare parameters and variables as attributes of the `declare` element. In addition, we define behavior such as ordering and result range as attributes of the `query` element.

### ***Example 11.26. Single-String Named Query***

Here is the same query as the previous example, using single-string form and implicit parameters and variables. Because the single-string form includes the candidate class, you can use it outside of a `class` element.

```
<?xml version="1.0"?>
<jdoquery>
  <query name="findBySub">
    <jdoql>
      select from org.mag.Magazine where articles.contains (art)
      &amp;&amp; art.subtitles.contains (:sub)
      order by title ascending range 0, 10
    </jdoql>
  </query>
</jdoquery>
```

### ***Example 11.27. Aggregates and Projections***

This example uses the `result` element to create an aggregate result. The query below returns each publisher and the average price of its magazines, for all publishers that publish less than 10 magazines. The results are packed into instances of the `PubPrice` class.

```
<?xml version="1.0"?>
<jdoquery>
  <package name="org.mag">
    <class name="Magazine">
      <query name="publishPrice">
        <result grouping="publisher having count (this) < 10" class="PubPrice">
          publisher, avg(price) as avgPrice
        </result>
      </query>
    </class>
  </package>
```

```
</jdoquery>
```

To obtain a named query you pass the query's candidate class (or null for queries outside of a `class` element) and name to the `PersistenceManager`. Once you have the `Query`, you can immediately execute it, optionally passing in values for any parameters the query declares.

### *Example 11.28. Executing Named Queries*

```
PersistenceManager pm = ...;
Query query = pm.newNamedQuery (Magazine.class, "findBySub");
Collection mags = (Collection) query.execute ("JDO");
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

## 11.11. Conclusion

---

The `Query` interface is a powerful weapon in your JDO arsenal. In this chapter, we focused on how to use queries in conjunction with the datastore-neutral, object-oriented JDO query language, JDOQL. The next chapter details how to partner the `Query` interface with SQL instead.

---

# Chapter 12. SQL Queries

JDOQL is a powerful query language, but there are times when it is not enough. Maybe you're migrating a JDBC application to JDO on a strict deadline, and you don't have time to translate your existing SQL selects to JDOQL. Or maybe a certain query requires database-specific SQL your JDO implementation doesn't support. Or maybe your DBA has spent hours crafting the perfect select statement for a query in your application's critical path. Whatever the reason, SQL queries can remain an essential part of an application.

You are probably familiar with executing SQL queries by obtaining a `java.sql.Connection`, using the JDBC APIs to create a `Statement`, and executing that `Statement` to obtain a `ResultSet`. And of course, you are free to continue using this low-level approach to SQL execution in your JDO applications. However, JDO also supports executing SQL queries through the `javax.jdo.Query` interface introduced in [Chapter 11, Query \[54\]](#). Using a JDO SQL query, you can retrieve either persistent objects or projections of column values. The following sections detail each use.

## Note

SQL queries are part of the JDO 2 specification. Though Kodo supports all of the features detailed in this chapter, many JDO 1 implementations may not. This chapter describes SQL queries as they appear in the JDO 2 Early Draft specification; their behavior may change before JDO 2 is finalized.

Kodo also supports embedding SQL into standard JDOQL queries. See [Section 10.7, “Query Extensions” \[290\]](#) in the Reference Guide for details.

SQL queries require Kodo JDO Enterprise Edition.

---

## 12.1. Creating SQL Queries

The `PersistenceManager` has two factory methods suitable for creating SQL queries:

```
public Query newQuery (Object query);
public Query newQuery (String language, Object query);
```

The first method is used to create a new query instance with the same properties as the passed-in query template. The template might be a query from another persistence manager, or a query that has been deserialized and has lost its persistence manager association. The method works for any query, regardless of the language used.

The second method was designed specifically for non-JDOQL queries. Its first parameter is the query language to use. For SQL queries, the language is `"javax.jdo.query.SQL"` (in case you are wondering, the official JDOQL language string is `"javax.jdo.query.JDOQL"`). Its second parameter represents the query to run -- in this case, the SQL string. The example below shows these methods in action.

### *Example 12.1. Creating a SQL Query*

```
PersistenceManager pm = ...;
Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT * FROM MAGAZINE");
query.setClass (Magazine.class);
processMagazines ((Collection) query.execute ());
query.closeAll ();

Query template = deserializeTemplateQuery ();
query = pm.newQuery (template);
processMagazines ((Collection) query.execute ());
query.closeAll ();
```

While JDOQL queries have separate result, filter, grouping, and ordering strings, a single `SELECT` statement encompasses a complete SQL query. Thus, most methods of `SQL Query` objects throw an exception. In particular, you cannot call the following methods:

- `setCandidates (Collection)`
- `setFilter (String)`
- `setResult (String)`
- `setGrouping (String)`
- `setOrdering (String)`
- `declareImports (String)`
- `declareVariables (String)`
- `declareParameters (String)`

### Note

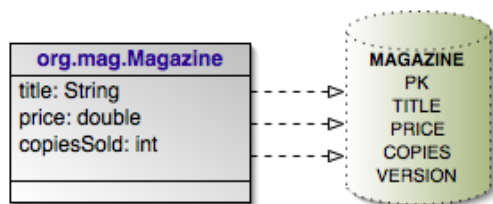
In addition to `SELECT` statements, Kodo supports stored procedure invocations as SQL queries. Kodo will assume any SQL that does not begin with the `SELECT` keyword (ignoring case) is a stored procedure call, and invoke it as such at the JDBC level.

## 12.2. Retrieving Persistent Objects with SQL

When you give a SQL query a candidate class, it will return persistent instances of that class. At a minimum, your SQL must select the class' primary key columns, class indicator column (if mapped), and version column (also if mapped). The JDO runtime uses the values of the primary key columns to construct each result object's identity, and possibly to match it with a persistent object already in the persistence manager's cache. When an object is not already cached, the implementation creates a new object to represent the current result row. It might use the class indicator column value to make sure it constructs an object of the correct subclass. Finally, the query records available version column data for use in optimistic concurrency checking, should you later change the result object and flush it back to the database.

Aside from the primary key, class indicator, and version indicator columns, any columns you select are used to populate the persistent fields of each result object. JDO implementations will compete on how effectively they map your selected data to your persistent instance fields.

Let's make the discussion above concrete with an example. It uses the following simple mapping between a class and the database:



### Example 12.2. Retrieving Persistent Objects

```
Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT PK, TITLE, PRICE, "
    + "VERSION FROM MAGAZINE WHERE PRICE > 5 AND PRICE < 10");
query.setClass (Magazine.class);
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (results);
```

It is very important to notice that we explicitly set the candidate class for the query. If we had not performed this step, the query would have been treated as a projection. We cover SQL projections later in this chapter.

The query above works as advertised, but isn't very flexible. Let's update it to take in parameters for the minimum and maximum price, so we can reuse it to find magazines in any price range:

### *Example 12.3. SQL Query Parameters*

```
Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT PK, TITLE, PRICE, "
    + "VERSION FROM MAGAZINE WHERE PRICE > ? AND PRICE < ?");
query.setClass (Magazine.class);

Double min = new Double (5D);
Double max = new Double (10D);
Collection results = (Collection) query.execute (min, max);
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (results);
```

Like JDBC prepared statements, SQL queries represent parameters with question marks. When you run a query with multiple parameters, the order of your arguments to the `execute` method must match the order of the question marks for each parameter in your SQL. To use the Query interface's `executeWithMap` method, imagine that the question marks are labeled from 1 to N, and key each parameter value on the correct Integer position.

## 12.3. SQL Projections

SQL queries without a candidate class are treated as projections of column data. If you select a single column, the query returns a collection of Objects. If you select multiple columns, it returns a collection of Object[]s. In either case, each column value is obtained using the `java.sql.ResultSet.getObject` method. The following example demonstrates a query for the values of the PK and VERSION columns of all MAGAZINE table records, using the data model we defined in [Section 12.2](#), **“Retrieving Persistent Objects with SQL”** [77].

### *Example 12.4. Column Projection*

```
Query query = pm.newQuery ("javax.jdo.query.SQL",
    "SELECT PK, VERSION FROM MAGAZINE");
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) results.next ();
    processPrimaryKeyAndVersion (data[0], data[1]);
}
query.close (results);
```

Notice that in the code above, we did not set a candidate class. Therefore, the query is treated as a projection.

Our discussion of JDOQL query result classes in **Section 11.8, “Result Class” [68]** also applies to SQL queries. As with JDOQL queries, SQL queries can automatically pack their results into objects of a specified type. JDO uses the `java.sql.ResultSetMetaData.getColumnLabel` method to match each column alias to the result class' public fields and JavaBean setter methods. Here is a modification of our example above that packs the selected column values into JavaBean instances.

### *Example 12.5. Result Class*

```
public class Identity
{
    private long id;
    private int versionNumber;

    public void setId (long id)
    {
        this.id = id;
    }

    public long getId ()
    {
        return id;
    }

    public void setVersionNumber (int versionNumber)
    {
        this.versionNumber = versionNumber;
    }

    public int getVersionNumber ()
    {
        return versionNumber;
    }
}

Query query = pm.newQuery ("javax.jdo.query.SQL",
    "SELECT PK AS id, VERSION AS versionNumber FROM MAGAZINE");
query.setResultClass (Identity.class);
Collection results = (Collection) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processIdentity ((Identity) itr.next ());
query.close (results);
```

## 12.4. Named SQL Queries

---

We discussed how to declare named JDOQL queries in **Section 11.10, “Named Queries” [72]**. Named queries, however, aren't limited to JDOQL. By replacing the `filter` attribute or element with `sql` in the XML, you can define a named SQL query. And because SQL queries do not require a candidate class, you can declare them outside of a `class` element.

### *Example 12.6. Named SQL Queries*

```
<?xml version="1.0"?>
<jdoquery>
  <query name="avgPrice">
    <sql>SELECT AVG(PRICE) FROM MAGAZINE</sql>
  </query>
  <package name="org.mag">
    <class name="Magazine">
      <query name="ttl" sql="SELECT * FROM MAGAZINE WHERE TITLE = ?">
        <result unique="true"/>
      </query>
    </class>
  </package>
</jdoquery>
```

The `avgPrice` query above returns the average price of all magazines. Because it is a SQL aggregate, it does not have a candidate class, and so we specify it at the root level.

The `t1` query returns the magazine with a the title given on execution. For this query, we chose to use the `sql` attribute rather than the element form used by the first query. The code below executes both queries.

```
PersistenceManager pm = ...;
Query query = pm.newNamedQuery (null, "avgPrice");
double avgPrice = ((Number) query.execute ().doubleValue ());

query = newNamedQuery (Magazine.class, "t1");
Magazine jdj = (Magazine) query.execute ("JDJ");
```

## 12.5. Conclusion

---

If you've used relational databases extensively, you might be tempted to perform all your JDO queries with SQL. Try to resist this temptation. SQL queries tie your application to the particulars of your current data model and database vendor. If you stick with JDOQL, on the other hand, you can port your application to other schemas and database vendors without any changes to your code. Additionally, most JDO implementations already produce highly optimized SQL from your JDOQL filters, and many are able to cache JDOQL query results for added performance.

---

# Chapter 13. Conclusion

This concludes our overview of the Java Data Objects specification. The **Kodo JDO Tutorials** continue your JDO education with step-by-step instructions for building simple JDO applications. Finally, the **Kodo JDO Reference Guide** contains detailed documentation on all aspects of SolarMetric's Kodo JDO implementation and development tools.

---

## **Part III. Kodo JDO Tutorials**

---

---

# Table of Contents

1. Kodo JDO Tutorials .....	84
1.1. Tutorial Requirements .....	84
2. Kodo JDO Tutorial .....	85
2.1. The Pet Shop .....	85
2.1.1. Included Files .....	85
2.1.2. Important Utilities .....	86
2.2. Getting Started .....	86
2.2.1. Configuring the Data Store .....	87
2.3. Inventory Maintenance .....	88
2.3.1. Persisting Objects .....	89
2.3.2. Deleting Objects .....	90
2.4. Inventory Growth .....	91
2.5. Behavioral Analysis .....	92
2.5.1. Complex Queries .....	96
2.6. Extra Features .....	97
3. Reverse Mapping Tool Tutorial .....	98
3.1. Magazine Shop .....	98
3.2. Setup .....	98
3.2.1. Tutorial Files .....	98
3.2.2. Important Utilities .....	99
3.3. Generating Persistent Classes .....	99
3.4. Using the Finder .....	101
4. J2EE Tutorial .....	103
4.1. Prerequisites for the Kodo J2EE Tutorial .....	103
4.2. J2EE Installation Types .....	103
4.3. Installing Kodo JCA .....	103
4.3.1. JBoss 3.0 .....	103
4.3.2. JBoss 3.2 .....	104
4.3.3. WebLogic 6.1 to 7.x .....	104
4.3.4. WebLogic 8.1 .....	105
4.3.5. WebSphere 5 .....	105
4.3.6. SunONE Application Server 7 / Sun Java Enterprise Server 8 - 8.1 .....	106
4.3.7. Macromedia JRun 4 .....	106
4.3.8. Borland Enterprise Server 5.2 - 6.0 .....	107
4.3.9. Non-JCA Application Server Deployment .....	107
4.4. Installing the J2EE Sample Application .....	108
4.4.1. Compiling and Building The Sample Application .....	109
4.4.2. Deploying Sample To JBoss .....	109
4.4.3. Deploying Sample To WebLogic 6.1 to 7.x .....	110
4.4.4. Deploying Sample To WebLogic 8.1 .....	110
4.4.5. Deploying Sample To SunONE / JES .....	110
4.4.6. Deploying Sample To JRun .....	110
4.4.7. Deploying Sample To WebSphere .....	110
4.4.8. Deploying Sample To Borland Enterprise Server 5.2 .....	111
4.5. Using The Sample Application .....	111
4.6. Sample Architecture .....	111
4.7. Code Notes and J2EE Tips .....	112

---

# Chapter 1. Kodo JDO Tutorials

These tutorials provide step-by-step examples of how to use various facets of the Kodo JDO system. They assume a general knowledge of JDO and Java. For more information on these subjects, see the following URLs:

- [Sun's Java site](#)
- [JDO Overview Document](#)
- [Links to JDO Resources](#)

## 1.1. Tutorial Requirements

---

These tutorials require 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 **Chapter 2, *SolarMetric Kodo JDO Installation* [4]** of Part I of this manual (the SolarMetric Kodo JDO README) or see the **README.txt** included in the root directory of your download for more information on requirements and installation procedures.

---

# Chapter 2. Kodo JDO Tutorial

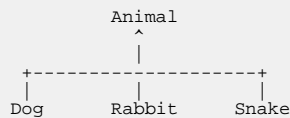
In this tutorial you will become familiar with the basic tools and development processes under Kodo by creating a simple JDO application.

## 2.1. 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:



### 2.1.1. 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.AnimalMaintenance:** Provides some utility methods for examining and manipulating the animals stored in the database. We will fill in method definitions in [Section 2.3, “Inventory Maintenance” \[88\]](#)
- **tutorial.Animal:** This 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 [Section 2.4, “Inventory Growth” \[?\]](#).
- **tutorial.Snake:** Contains data and methods specific to snakes. It will be used in [Section 2.5, “Behavioral Analysis” \[?\]](#).
- **package.jdo:** This 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 [Chapter 5, Metadata \[27\]](#) of the JDO Overview.
- **kodo.properties:** This properties file contains Kodo-specific and standard JDO configuration settings.

#### Important

You must specify a valid Kodo license key in the `kodo.properties` file.

- **solutions:** The solutions directory contains the complete solutions to this tutorial, including finished versions of the `.java` files listed above and a correct `package.jdo` metadata file.

## 2.1.2. Important Utilities

---

- **java**: Runs main methods in specified Java classes.
- **javac**: Compiles `.java` files into `.class` files that can be executed by **java**.
- **jdoc**: Runs the Kodo JDO enhancer against the specified classes. More information is available in [Section 5.5, “Enhancement” \[192\]](#) of the Reference Guide.
- **mappingtool -action refresh**: A utility that can be used to create and maintain the object-relational mappings and schema of all persistent classes in a JDBC-compliant data store. This functionality allows the underlying mappings and schema to be easily kept up-to-date with the Java classes in the system. See [Chapter 7, Object-Relational Mapping \[209\]](#) of the Reference Guide for more information.

## 2.2. 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:

### Note

Be sure that your `CLASSPATH` is set correctly. Please see [Section 2.9, “Windows Installation \(no installer -- zip file\)” \[5\]](#) of Part I of this manual for Windows, or [Section 2.10, “POSIX \(Linux, Solaris, Mac OS X, Windows with cygwin, etc.\) Installation” \[6\]](#) of Part I of this manual for POSIX (Linux, Solaris, Cygwin, etc.). Detailed information on which libraries are needed can be found in [Appendix G, Development and Runtime Libraries \[430\]](#). Note, also, that your Kodo install directory should be in the `CLASSPATH`, as the tutorial classes are located in the `tutorial` directory under your Kodo install directory, and are in the `tutorial` package.

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 package.jdo
```

This step runs the Kodo JDO enhancer on the `package.jdo` file mentioned above. The `package.jdo` file contains an enumeration of all the classes that should be JDO enhanced. The Kodo JDO enhancer will examine the metadata defined in this file and enhance all classes listed in it appropriately. See [Section 5.5, “Enhancement” \[192\]](#) of the Reference Guide for more information on the JDO enhancer.

## 2.2.1. Configuring the Data Store

---

Now that we've compiled the source files and enhanced the JDO classes, we're ready to set up the database. **Hypersonic SQL**, a pure Java relational database, is included in this distribution. 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 only describes how to set up connectivity 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 [Chapter 4, JDBC \[166\]](#) of the Reference Guide.

1. Create the object-relational mappings and database schema.

```
mappingtool -action refresh package.jdo
```

This command creates object-relational mappings for the classes listed in `package.jdo`, and at the same time propagates the necessary schema to the database configured in `kodo.properties`. If you are using the default Hypersonic SQL setup, the first time you run the mapping 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.

By default, Kodo stores object-relational mapping information in `.mapping` files. As you will see in the [Reverse Mapping Tool Tutorial](#), you can also configure Kodo to store object-relational mappings in your JDO metadata files or in a database table. [Chapter 7, Object-Relational Mapping \[209\]](#) of the Reference Guide describes your mapping options in detail.

If you'd like to see the mapping information Kodo has just created for the classes listed in `package.jdo`, examine the `package.mapping` file. Again, [Chapter 7, Object-Relational Mapping \[209\]](#) of the Reference Guide will help you understand mapping XML in detail, should the need ever arise. Most Kodo development does not require any knowledge of mappings.

If you are curious, you can also view the schema Kodo created for the tutorial classes with Kodo's schema generator tool:

```
schemagen -file tmp.schema
```

This will create a `tmp.schema` file with an XML representation of the database schema. The XML should be self explanatory; see [Section 8.3, “XML Schema Format” \[282\]](#) of the Reference Guide for details. You may delete the `tmp.schema` file before proceeding.

2. Populate the database with sample data.

```
java tutorial.SeedDatabase
```

Congratulations! You have now created a JDO-accessible persistent store, and seeded it with some sample data.

## 2.3. 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 mapping tool 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 [Chapter 11, Query \[54\]](#) of 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.

## 2.3.1. Persisting Objects

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.

`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 PersistenceManagerFactory and PersistenceManager.
    PersistenceManagerFactory pmf =
        KodoHelper.getPersistenceManagerFactory ("kodo.properties");
    PersistenceManager pm = pmf.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 and PersistenceManagerFactory.
    pm.close ();
    pmf.close ();
}
```

### Note

In the above code, we used the `kodo.runtime.KodoHelper` class. This class is similar to `javax.jdo.JDOHelper`, except that it provides some convenience methods for obtaining a `PersistenceManagerFactory` from properties enumerated in a resource at a given location, or in a specific JNDI location. This code could be implemented in a fully standards-based manner by loading the resource named `kodo.properties` into a `Properties` object and then passing this object to `JDOHelper.getPersistenceManagerFactory`.

Also note that equivalent convenience methods are being considered by the JDO expert group for inclusion in the JDO 2 standard.

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
```

You can view the contents of the database with:

```
java tutorial.AnimalMaintenance list Dog
```

---

## 2.3.2. Deleting Objects

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 delete with
 */
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.

## 2.4. 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 `package.jdo`.

The `Rabbit` class above contains a two-sided many-to-many relationship, between parents and children. From the Java side of things, a two-sided 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 mapping tool to create the most efficient schema. The snippet below should be inserted into the `package.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 uses the JDO extension mechanism to add inverse metadata to the JDO metadata file. For more information on metadata, consult **Chapter 6, Metadata [197]** of the Kodo JDO Reference Guide.

Add the following code immediately *before* the `</package>` line in the `package.jdo` file.

```
<class name="Rabbit" persistence-capable-superclass="Animal" >
  <field name="parents">
    <collection element-type="Rabbit"/>
  </field>
  <field name="children">
    <collection element-type="Rabbit"/>
    <extension vendor-name="kodo" key="inverse-owner" value="parents"/>
  </field>
</class>
```

```
</field>
</class>
```

3. Enhance the Rabbit class.

```
jdoc Rabbit.java
```

4. Refresh the object-relational mappings and database schema.

```
mappingtool -action refresh Rabbit.java
```

Now that we have a Rabbit class, let's get some preliminary rabbit data into the database.

1. Create some rabbits.

Run the following commands a few times to add some male and female rabbits to the database:

```
java tutorial.AnimalMaintenance add Rabbit <name> false
java tutorial.AnimalMaintenance add Rabbit <name> true
```

Now run some breeding iterations.

```
java tutorial.Rabbit breed 2
```

2. Look at your new rabbits.

```
java tutorial.AnimalMaintenance list Rabbit
java tutorial.AnimalMaintenance details Rabbit ""
```

---

## 2.5. 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 `package.jdo`.

```
<class name="Snake" persistence-capable-superclass="Animal"/>
```

3. Enhance the class.

```
jdoc Snake.java
```

4. Refresh the mappings and database.

As we have created a new persistence-capable class, we must map it to the database and change the schema to match. So run the mapping tool:

```
mappingtool -action refresh Snake.java
```

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. To see the new snakes in the database, run:

```
java tutorial.AnimalMaintenance list Snake
```

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)
{
    PersistenceManagerFactory pmf =
        KodoHelper.getPersistenceManagerFactory ("kodo.properties");
    PersistenceManager pm = pmf.getPersistenceManager ();
    Transaction transaction = pm.currentTransaction ();
    transaction.begin ();

    // 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");

        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.");
                break;
            }
            else
            {
                // 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);
            }
        }
    }
}
```

```

        }
    }
    else
        System.out.println ("No snakes matching '" + filter
            + "' found in persistence manager");

    transaction.commit ();
    pm.close ();
    pmf.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 package.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 package.jdo file.

```

<class name="Snake" persistence-capable-superclass="Animal" >
  <field name="giTract">
    <collection element-type="Rabbit"/>
    <extension vendor-name="kodo" key="inverse-owner" value="eater"/>
  </field>
</class>

```

Note that we specified an `inverse-owner` 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 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 **Chapter 7, *Object-Relational Mapping* [209]** of the Kodo JDO Reference Guide.

4. Compile `Snake.java` and `Rabbit.java` and jdo-enhance the classes.

```
javac Snake.java Rabbit.java
jdoc Snake.java Rabbit.java
```

5. Refresh the mappings and database.

```
mappingtool -action refresh Snake.java Rabbit.java
```

Now, experiment with the following commands:

```
java tutorial.Snake eat
java tutorial.AnimalMaintenance details Snake ""
```

## 2.5.1. 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 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:

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

Note the use of the `animal` variable that was registered with the query. To add support for this variable, uncomment the commented-out `declareVariables` call in the `getAnimals` method in `AnimalMaintenance.java`:

1. `AnimalMaintenance.getAnimals` should look like this:

```
/**
```

```
* Return a collection of animals that match the specified query filter.
*
* @param filter    the JDO filter to apply to the query
* @param cls       the class of animal to query on
* @param pm        the PersistenceManager to obtain the query from
*/
public static Collection getAnimals (String filter, Class cls,
    PersistenceManager pm)
{
    // Get a query for the specified class and filter.
    Query query = pm.newQuery (cls, filter);

    // Add a single variable of type 'Animal' to this query to allow
    // for some reasonably powerful queries. This will be uncommented
    // in Chapter V.
    query.declareVariables ("Animal animal;");

    // Execute the query.
    return (Collection) query.execute ();
}
```

## 2. Recompile AnimalMaintenance.java.

```
javac AnimalMaintenance.java
```

The `animal` variable is now available to use to represent any `Animal` contained in a collection. As `animal` was declared as type `Animal`, we cannot directly use fields in our `Animal` subclasses. However, the JDO specification provides support for casting in queries, which lets us run queries like:

```
java tutorial.AnimalMaintenance details Snake "giTract.contains (animal) && ((Rabbit) animal).isFemale == false"
```

This prints details about all snakes that have eaten male rabbits.

## 2.6. 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.

---

# Chapter 3. Reverse Mapping Tool Tutorial

In this tutorial you will learn how to use Kodo JDO's reverse mapping tool to reverse-engineer persistent classes from a database schema.

## 3.1. 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.

## 3.2. 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 reverse-tutorial directory.

### 3.2.1. Tutorial Files

---

The tutorial uses the following files:

- The `reversetutorial_database.properties`, `reversetutorial_database.script`, and `reverse-`

`tutorial_database.data` : These files make up a Hypersonic file-based database with the schema outlined above. The database is already populated with lots of magazine data representing your shop's inventory.

- **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 values below:

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

### 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.

## 3.2.2. Important Utilities ---

- **java**: Runs main methods in specified Java classes.
- **javac**: Compiles `.java` files into `.class` files that can be executed by **java**.
- **jdoc**: Runs the Kodo JDO enhancer against the specified classes. More information is available in [Section 5.5](#), “**Enhancement**” [192]f the Reference Guide.

## 3.3. 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 section.
2. Now that we have 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 `schemagen` shell script. The `-file` flag tells the generator what to name the XML file it creates:

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

The schema generator is described in detail in [Section 8.1.3, “Schema Generator” \[279\]](#) of the Reference Guide.

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 in [Section 8.3, “XML Schema Format” \[282\]](#) of the Reference Guide.
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 `reversemappingtool` script. Use the `-package` flag to control the package name of the generated classes.

```
reversemappingtool -package reversetutorial schema.xml
```

The reverse mapping tool has many options and customization hooks. For a complete treatment of the tool, see [Section 5.6, “Auto-Generating Classes from a Schema” \[193\]](#) of the Reference Guide.

Running the reverse mapping 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 `package.mapping` file. The mapping file contains the object-relational information linking the generated classes to your existing schema. [Section 7.4, “Mapping File XML Format” \[215\]](#) of the Reference Guide discusses the mapping file format in detail.

5. The `.mapping` file generated by the reverse mapping tool is suitable for use with the default **file mapping factory**. Kodo JDO offers other factories that store mapping data in other formats.

As shop owner, you've decided that you don't want an extra `.mapping` file lying around like you have now. Instead, you'd like to store the mapping information in the JDO metadata file, using metadata's built-in extension mechanism. Kodo JDO supports storing object-relational mappings in JDO metadata with the **metadata mapping factory**. To use this factory, add the following line to the `kodo.properties` file:

```
kodo.jdbc.MappingFactory: metadata
```

6. Whenever you switch mapping factories, you have to import the `.mapping` file data into the new factory. We accomplish this with the **mapping tool**. Make sure to compile the generated classes before the import:

```
javac *.java
mappingtool -action import package.mapping
```

You can now delete the mapping file if desired.

7. 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 `package.jdo` metadata file. It contains the necessary standard JDO metadata, plus, thanks to our use of the mapping tool's import action, 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 one additional step before you can use the classes with persistence: enhancement.

```
jdoc package.jdo
```

This step runs the Kodo JDO enhancer on the `package.jdo` file mentioned above. The `package.jdo` file contains an enumeration of all the classes that should be JDO enhanced. The Kodo JDO enhancer will examine the file's metadata and enhance all listed classes appropriately. See [Section 5.5, “Enhancement” \[192\]](#) in the Reference Guide for more information on the JDO enhancer.

Congratulations! You are now ready to use JDO to access your magazine data.

## 3.4. 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 [Chapter 11, Query \[54\]](#) of 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 "Kodo" in any of its magazines:

```
java reversetutorial.Finder "publisher.magazines.contains (mag) && mag.articles.contains (art) && art.articleSubtitles.contains (
```

Have fun experimenting with additional queries.

### Note

If you plan on using Kodo JDO to modify data in a schema with non-deferred foreign key constraints, make sure to add the following line to `kodo.properties`:

```
kodo.jdbc.ForeignKeyConstraints: true
```

This line tells Kodo JDO to order its SQL to meet the foreign key constraints of your schema. Note that though this tutorial uses foreign keys with the included Hypersonic database, SolarMetric recommends that you do not use foreign keys with Hypersonic in general use. In fact, in their default configurations the Kodo tools will refuse to create foreign keys in a Hypersonic database. This is because foreign keys seem to destabilize the database under heavy load.

Also, make sure to erase the `kodo.jdbc.MappingFactory` setting in `kodo.properties` before continuing with your own JDO experimentation, unless you'd like to always store your mapping information in your JDO metadata files rather than the database.

---

# Chapter 4. J2EE Tutorial

By deploying Kodo into a J2EE environment, you 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 and showcase some basic enterprise JDO design techniques. The tutorial's sample application attempts to model a basic garage catalog system. While the application is relatively trivial, the code has been constructed to illustrate simple patterns and solutions to common problems when using Kodo JDO in an enterprise environment.

## 4.1. 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, JRun, or SunONE / Sun JES. If you use a different application server, 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.

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.

## 4.2. J2EE Installation Types

---

Every application server has a different installation process for installing J2EE components. Kodo can be installed in a number of ways, each of 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 a 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 a 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.
- **JDBCPersistenceManagerFactory:** This remains the most compatible way to integrate Kodo into a J2EE environment, though this is not seamless and can require a fair bit of custom application server code to manually bind an instance of `JDBCPersistenceManagerFactory` 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.

## 4.3. Installing Kodo JCA

---

### 4.3.1. JBoss 3.0

---

The `jca` directory of the Kodo distribution includes `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 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`). Then copy the `jdo-1.0.2.jar` file to the `lib` directory of the JBoss server installa-

tion (e.g., `jboss-3.0.6/server/lib/`), so that the common JDO APIs are available globally in the JBoss installation. 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 `kodo.jdbc.ee.JDBCConnectionFactory` bound at the JNDI name you specified. You have now installed Kodo JCA.

## 4.3.2. 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. `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.

## 4.3.3. WebLogic 6.1 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 6.1.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. Make sure to also add the JDO base jar (`jdo-1.0.2.jar`) to the classpath so that your JDO classes can be loaded. While this jar 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, select `mydomain/Deployments/Connectors` in the left navigation bar and then select "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 propagate them to the active domain configuration.

You should see WebLogic attempt to deploy Kodo in the system console. When it is done, 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.

### Note

When using WebLogic 7, you may get invalid DTD exceptions when loading JDO metadata files, due to a problem with loading resources that are in jar files inside a resource archive. You can work around these problems by specifying a non-public DTD in your metadata files, like so:

```
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">
```

## 4.3.4. 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 `jdo-1.0.2.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.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 and 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.

## 4.3.5. WebSphere 5

---

Websphere installation is easiest through 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.

## 4.3.6. SunONE Application Server 7 / Sun Java Enterprise Server 8 -

---

## 8.1

---

Installation in SunONE / JES 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.

```
permission javax.jdo.spi.JDOPermission ".*";
```

Now restart SunONE / JES.

SunONE / JES requires a SunONE / JES-specific deployment file in addition to the generic `ra.xml`. Edit the `sun-ra.xml` file provided in the `jca` directory of your Kodo installation by setting `jndi-name` attribute to the JNDI name of your choice. Then 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` in 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 / JES 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, SunONE / JES Application Server sometimes may not accept its own configuration file format. If this is the case, you will see 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.

## 4.3.7. Macromedia JRun 4

---

JRun requires a JRun-specific deployment file in addition to the generic `ra.xml`. Edit the `jrun-ra.xml` file provided in the `jca` directory of your Kodo installation by setting `jndi-name` attribute to the JNDI name of your choice. Then 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 `jrun-ra.xml` in the `META-INF` virtual directory in the archive:

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

Browse to the web admin console of JRun in your browser. Select your server in the servers tree and expand to `J2EE Components`. Under `Resource Adapters`, select `Add` and upload your new `kodo.rar` file. The Kodo resource adapter will now be deployed to the JNDI name that you specified in the `jrun-ra.xml` file.

### Note

JRun does not provide any means of configuring a built-in DataSource that is not enlisted in a global transaction. This means that when configuring Kodo to use an external DataSource bound into JRun, you must also configure Kodo's `Connection2URL`, `Connection2DriverName`, `Connection2UserName`, and `Connection2Password` properties in order to provide Kodo with a nontransactional connection.

## 4.3.8. Borland Enterprise Server 5.2 - 6.0

---

### Note

Borland includes two different `TransactionManagers` for controlling the J2EE environment depending on your configuration. Kodo supports the standard instance on all versions of Borland. However, to use the two phase commit `TransactionManager`, labeled as the OTS service, you must use version 6.0 or higher. This is needed to support true XA transactions (prior versions had limited Java availability).

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. Move `jdo-1.0.2.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 `jdo-1.0.2.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.

## 4.3.9. Non-JCA Application Server Deployment

---

For application servers that do not support JCA as a means of deploying resource adapters, Kodo can be deployed manually by writing some code to configure and bind an instance of the `PersistenceManagerFactory` into JNDI. The mechanism by which you do this is up to you and will be dependant on functionality that is specific to your application server. The most common way is to create a startup class according to your application server's documentation that will create a factory and then bind it in JNDI. For example, in WebLogic you could write a startup class as follows:

**Example 4.1. Binding a PersistenceManagerFactory into JNDI via a WebLogic Startup Class**

```

import java.util.*;
import javax.naming.*;

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import weblogic.jndi.WLContext;

/**
 * This startup class creates and binds an instance of a
 * Kodo JDBCPersistenceManagerFactory into JNDI.
 */
public class StartKodo
    implements T3StartupDef
{
    private static final String PMF_JNDI_NAME = "my.jndi.name";
    private static final String PMF_PROPERTY =
        "javax.jdo.PersistenceManagerFactoryClass";
    private static final String PMF_CLASS_NAME =
        "kodo.jdbc.runtime.JDBCPersistenceManagerFactory";

    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 = PMF_JNDI_NAME;

        Properties props = new Properties ();
        props.setProperty (PMF_PROPERTY, PMF_CLASS_NAME);

        // you could set additional properties here; otherwise, the defaults
        // from kodo.properties will be used (if the file exists)

        PersistenceManagerFactory factory = JDOHelper.
            getPersistenceManagerFactory (props);

        Hashtable icprops = new Hashtable ();
        icprops.put (WLContext.REPLICATE_BINDINGS, "false");
        InitialContext ic = new InitialContext (icprops);
        ic.bind (jndi, factory);

        // return a message for logging
        return "Bound PersistenceManagerFactory to " + jndi;
    }
}

```

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

**Example 4.2. Looking up the PersistenceManagerFactory in JNDI**

```

PersistenceManagerFactory factory = (PersistenceManagerFactory)
    new InitialContext ().lookup ("java:/MyKodoJNDIName");
PersistenceManager pm = factory.getPersistenceManager ();

```

## 4.4. 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.

### 4.4.1. 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 nested `ejb` directory:

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

Enhance the Car class.

```
jdoc package.jdo
```

Run the mapping tool; make sure that your `kodo.properties` file includes the same connection information (e.g. Driver, URL, etc.) as your installation:

```
mappingtool -a 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.

JBoss, for example, 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
```

### 4.4.2. Deploying Sample To JBoss

---

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 JNDI.

---

### 4.4.3. Deploying Sample To WebLogic 6.1 to 7.x

---

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` .

### 4.4.4. 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 -> 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` .

### 4.4.5. Deploying Sample To SunONE / JES

---

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.

### 4.4.6. Deploying Sample To JRun

---

Browse to the admin console in your web browser. Select `J2EE Components` from the left navigation panel. Select `Add` under the `Enterprise Applications` heading. Select the `samplej2ee.ear` file and hit `Deploy`. You should now see `Sample-KodoJ2EE` listed in the top-level folder of the navigation panel. Select it, and then select the `sample-ejb.jar#CarEJB` component under `Enterprise JavaBeans` section, then change the `JNDI Name` for the bean from the default value to `samples.j2ee.ejb.CarHome` and hit `Apply`.

If the Kodo resource adapter and the sample EAR are both configured correctly, you should now be able to access your sample application at JRun's deploy URL (e.g., "`http://localhost:8100/sample/`").

### 4.4.7. 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.

## 4.4.8. Deploying Sample To Borland Enterprise Server 5.2

---

Deploy the EAR file using iastool or the console. Note that you may have to include the JDO library in your stub generation process. Also be sure that you have followed the JCA instructions for BES as well as editing the samples.properties to point to serial://kodo JNDI location. You should be able to see the CarEJB located in JNDI located at the home classname.

## 4.5. 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"`).

## 4.6. Sample Architecture

---

The garage 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 session bean facade around a persistence capable instance. Using a session bean 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 session bean 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 entity beans. See our sample ejb directory and JDOEntityBean for more details on how to using JDO to power your entity beans.

- `samples/j2ee/Car.java`: The core of the sample application. This is the persistence capable 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, Car implements `java.io.Serializable` so that remote clients can access cars 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. These JSPs are not aware of JDO; they simply 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.

## 4.7. Code Notes and J2EE Tips

---

1. JDO persistence capable instances are excellent candidates for 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 you 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 encapsulate complex business logic at the server level.

2. Persistent instances using datastore identity lose all identity upon serialization (which happens when they are 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 stringified 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 persistent instances under application-identity still lose their id instance, recreating it from the server side is 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);
```

- Wrapper objects that store the persistent 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
}
```

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 persistence manager, it allows the JDO implementation to free up unused resources. Since a persistence manager is created for every transaction, this can increase the scalability of your application.
4. You should not use `PersistenceManager.currentTransaction` or any `javax.jdo.Transaction` methods.

Instead, use the JTA transactional API. `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 straightforward, 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 before being returned from the server.
  - While "default fetch group" values will always be returned to the client upon serialization, lazily loaded fields will not as the persistence manager will have been closed before those fields attempt to serialize. One can either simply access those fields before serialization, or one can use the persistence manager's `retrieve` and `retrieveAll` methods to make sure object state is loaded. Note that these methods are not recursive. If you need to go through multiple relations, you must call `retrieve` at each relational depth. This is an intentional limitation in the specification to prevent the entire object graph from being serialized and/or retrieved.
6. It is not necessarily required that you use EJBs and container-managed transactions to demarcate transactions, although that is probably the most common method. In EJBs using bean managed transactions, you can control transactions through either the `javax.jdo.Transaction` or the `javax.transaction.UserTransaction`. Furthermore, outside of EJBs you can access the JDO layer with either transactional API.
7. The Kodo distribution includes source code for 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`, and handle common EJB interface methods such as `setEntityContext`. To use these classes, we recommend placing Kodo's jars into the system classpath and not into the ear. Ear deployment can cause classloader problems due to the multiple locations that these classes could be loaded from.
8. Persistence managers are allocated on a per-Transaction basis. Calling `getPersistenceManager` from the same persistence manager factory within the same EJB method call will always return the same instance.

```
SampleUtilities.getPersistenceManager ()  
== SampleUtilities.getPersistenceManager (); // will always be true
```

---

## **Part IV. Kodo JDO Frequently Asked Questions**

---

---

# Table of Contents

- 1. Kodo JDO Frequently Asked Questions ..... 116
  - 1.1. General ..... 116
  - 1.2. Database ..... 117
  - 1.3. Programming with Kodo ..... 119
  - 1.4. How do I ... ? ..... 122
  - 1.5. Common errors ..... 124
  - 1.6. Productivity tools ..... 124
  - 1.7. Performance ..... 125
  - 1.8. Scalability ..... 126
  - 1.9. Application servers ..... 126
  - 1.10. Locking ..... 127
  - 1.11. Transactions ..... 128

---

# Chapter 1. Kodo JDO Frequently Asked Questions

## 1.1. General

---

1.1.1. What is Kodo?

Kodo is an implementation of the Java Data Objects (JDO) standard that enables developers to transparently access persistent data stores via the Java programming language.

1.1.2. What is JDO?

JDO stands for Java Data Objects, and is a standard written by Sun Microsystems to provide transparent access to a variety of datastores, from relational databases to object databases to plain files. A good introduction to JDO can be found at **Part II, “Java Data Objects” [8]**.

1.1.3. Is Kodo a database?

No. Kodo provides a means to access an existing database.

1.1.4. Is Kodo an application server?

No, although Kodo can integrate seamlessly with any J2EE 1.3 compliant application server.

1.1.5. Does Kodo require an application server?

No. Kodo can be run without any external managed environment, although it can also be used from within an EJB container, a servlet, or any other managed environment that is J2EE compliant.

1.1.6. What is the difference between JDO and JDBC?

Java Database Connectivity (JDBC) is an API that allows developers to directly access a relational database. JDO is a data-store-agnostic approach that aims to reduce the complexity of designing persistent applications, and is not constrained to any particular type of datastore. Kodo JDO utilizes JDBC to access the relational database.

1.1.7. What is the difference between JDO and EJB?

Enterprise Java Beans are managed distributed components that handle application-level security and automatic transaction demarcation. In contrast, JDO simply provides a transparent means to access a datastore. EJB and JDO are complimentary technologies; developers can write their EJBs to utilize the transparent persistence provided by JDO, rather than being limited to the restrictions of using the built-in CMP persistence or vendor-specific application server extensions.

- 1.1.8.  
Do I need to know SQL to use Kodo?

No. Kodo JDO completely shields the developer from needing to write or debug SQL statements, although it does provide advanced extensions for developers who are familiar with SQL to create new mappings.

- 1.1.9.  
What standards does Kodo conform to?

Kodo conforms to the Java Data Objects 1.0.1 specification. Additionally, various parts of Kodo conform to other standards and specifications, including XML, JTA, JCA, JNDI, JDBC, EJB, JMX, XA, and J2EE.

- 1.1.1  
0. What version of Java does Kodo require?

Kodo requires JDK 1.2.2 or higher to run. Additionally, Kodo provides some additional features for Java 5 development, including support for persisting enum fields and full use of generics information (if present) in place of JDO metadata for collection and map element type data. These are only accessible when using Kodo in conjunction with a Java 5 runtime environment.

- 1.1.1  
1. I have problems or questions about Kodo. Where can I go for help?

The SolarMetric developer community can be accessed from <http://solarmetric.com/Support/Newsgroups>. Other support resources can be accessed at <http://solarmetric.com/Support>. Also, if you have a maintenance and technical support contract with SolarMetric, you can e-mail questions to [jdosupport@solarmetric.com](mailto:jdosupport@solarmetric.com).

- 1.1.1  
2. I think I found a bug in Kodo. Where do I report it?

The first thing you should do is search for any existing bugs in SolarMetric's bug tracking system: <http://bugzilla.solarmetric.com>. It is often the case that someone else may have already reported the bug, and a possible solution or workaround can be found in the existing bug report. If you are confident that your issue has not already been reported, you can report it to SolarMetric by posting on the community newsgroups or sending mail to [jdosupport@solarmetric.com](mailto:jdosupport@solarmetric.com).

## 1.2. Database

---

- 1.2.1.  
Does Kodo require a database to function?

Kodo does require an existing database against which to operate. However, Kodo ships with a small, open-source, pure-java database called Hypersonic that can be used for development without requiring an existing database installation.

- 1.2.2.  
What databases does Kodo support?

Kodo has built-in support for all the major databases, including Oracle, Microsoft SQL Server, Sybase, and Informix. In addition, Kodo provides APIs that allow the developer to adapt Kodo to work with any other database that provides a JDBC-compliant driver. A full list of the supported databases can be found at [Appendix B, Supported Databases \[404\]](#)

- 1.2.3. What additional software do I need to use Kodo with my database?

Kodo requires Java Development Kit version 1.2.2 or higher. Additionally, if you will be accessing a database other than Hypersonic, Kodo requires a JDBC driver that can communicate with your database. All other libraries needed by Kodo are provided in the Kodo distribution. A list of all the libraries that Kodo requires and uses can be found at **Appendix G, *Development and Runtime Libraries* [430]**

- 1.2.4. Where can I find the JDBC driver for my application?

JDBC drivers are typically obtained from the database vendor's web site. In addition, there are many third-party companies that provide JDBC drivers for various databases. For a comprehensive list of existing JDBC drivers, see the Sun JDBC driver database at: <http://servlet.java.sun.com/products/jdbc/drivers>.

- 1.2.5. What is the best JDBC driver to use for my application?

SolarMetric does not usually endorse any specific JDBC driver for a particular database. Provided the driver is truly JDBC compliant, it should work with Kodo without problems. Typically it is a good idea to first look at the JDBC driver that your database vendor provides, since these are often high-quality and free. An advantage of the transparent JDO API is that you can simply "drop in" a different JDBC driver version and change a few properties, and you can test your application without changing any code. This makes the process of profiling your application's performance with different JDBC drivers a very simple task.

- 1.2.6. What version of JDBC does Kodo use?

Kodo utilizes version 2.0 of the JDBC specification, and requires that a driver be JDBC 2.0 compliant. In addition, Kodo uses the JDBC 2 Standard Extensions, which are provided in the Kodo distribution.

- 1.2.7. Can Kodo integrate with a legacy database schema?

Certainly. Kodo provides a very flexible set of mapping options to be able to integrate your Java object model with almost any relational database schema. In addition, Kodo provides extensible APIs that allow the developer to create their own mappings for those non-standard relational constructs that may not be included with Kodo "out of the box". Furthermore, Kodo provides a reverse mapping tool, which allows developers to automatically generate a Java object model from an existing schema, which dramatically reduces the amount of time the developer needs to spend manually setting up the mappings. To get started with the reverse mapping tool, see **Chapter 3, *Reverse Mapping Tool Tutorial* [98]**.

- 1.2.8. I am designing a persistent model from scratch and don't want to deal with creating a database schema. Can Kodo generate a schema for me?

Yes. Kodo allows you to design your application in a object-centric fashion, and can automatically generate a consistent relational schema from your object model. This is ideal for developers who are designing a new application that does not need to integrate with an existing schema. See **Section 7.1, "Mapping Tool" [209]**

- 1.2.9. I have both an existing object model and an existing database schema. Can I use Kodo without making changes to either?

Probably. Kodo's mapping capabilities are quite flexible, and the Kodo mapping tools provide facilities to deal with this "meet-in-the-middle" scenario. See **Section 7.1.1, "Using the Mapping Tool" [210]** for a more detailed description.

- 1.2.1  
0. Can Kodo generate the DDL for my database?

Yes. Kodo can either issue the DDL directly in order to create or update your database, or it can create a SQL script file. This can be done from the command line or using Ant. See [Section 7.1.2, “Generating DDL SQL” \[212\]](#) for examples.

- 1.2.1  
1. Can a Kodo run against multiple databases?

Yes. Kodo can operate simultaneously against an arbitrary number of databases at the same time. Utilizing XA transactions, you can even ensure transactional consistency across a heterogeneous set of databases when using container managed transactions.

- 1.2.1  
2. Is there any limit to the number of rows Kodo can handle?

There is no limit to the size of table against Kodo can operate, nor is there any limit to the number of tables in the database. Kodo is used in applications that use databases ranging from just a few tables, to databases that have thousands of tables with millions or rows.

- 1.2.1  
3. Can Kodo work with foreign keys?

Kodo can work with both deferred and non-deferred constraints. With non-deferred constraints, Kodo can be configured to analyze foreign key dependencies before executing SQL. With deferrable constraints, you should either make deferred the default or extend the `DBDictionary` to set deferred as the default for every connection. To enable Kodo's foreign key analysis, see [Section 2.6.50, “`kodo.jdbc.ForeignKeyConstraints`” \[157\]](#)

- 1.2.1  
4. How can I change what schema Kodo is using?

Kodo can be configured to look at a subset of schemas, assign a default schema to mapping information, and to ignore the schema altogether. The first is controlled by the [Section 2.6.56, “`kodo.jdbc.Schemas`” \[158\]](#) option which enumerates the schemas to analyze. The other options are configured at the `DBDictionary` level using the `UseSchemaName` and `DefaultSchemaName` properties (see [Section 4.3, “Database Support” \[169\]](#))

- 1.2.1  
5. What do I have to do when using an external datasource?

While Kodo can handle external data sources, there are certain additional configuration options to be aware of. If your external data source automatically integrates with the global transaction (such as XA data sources), you should ensure that Kodo has access to a non-transactional datasource: [Section 4.2.1, “Enlisted Data Sources” \[169\]](#)

- 1.2.1  
6. How has Schema Tool changed since version 2.x?

Schema Tool has been redesigned to only manage the schema through XML representations of the schema. Mapping Tool now manages the mapping between your classes and database. Mapping Tool can optionally use Schema Tool to synchronize the database schema to create any missing database objects. For further information on Mapping Tool, see [Section 7.1, “Mapping Tool” \[209\]](#)

---

## 1.3. Programming with Kodo

- 1.3.1.  
How long will it take me to learn the JDO APIs?

The JDO API is designed to be extremely simple. You can view the entire JDO API at <http://java.sun.com/products/jdo/javadocs>.

- 1.3.2.  
What standard APIs do I need to be familiar with to use Kodo?

Aside from the JDO APIs, you do not need to have any expertise in any APIs aside from the basic standard ones that the Java core library provides.

- 1.3.3.  
What is the fastest way to get going with Kodo?

The Kodo tutorial provides a good introduction to developing a small application using the JDO APIs. See **Chapter 2, *Kodo JDO Tutorial* [85]**.

- 1.3.4.  
Do you provide any example applications using Kodo?

Kodo ships with numerous sample applications that can be adapted for your needs. For an overview of the samples that come with the Kodo distribution, see **Chapter 1, *Kodo Sample Code* [369]**

- 1.3.5.  
How do I issue queries to the database using Kodo?

JDO specifies a query language called JDOQL, which allows queries to be written in a object-centric way. See **Section 11.2, “JDOQL” [56]**. Kodo also offers the ability to directly execute SQL queries against the database. See **Section 13.4, “Direct SQL Execution” [324]**

- 1.3.6.  
How much more programming do I need to do to use Kodo?

One of the goals of JDO's transparent persistence is to make the persistence code in your application as minimal and unintrusive as possible. Typically, you will only need to write in the transaction demarcation, object queries, and the addition of root objects to the database.

- 1.3.7.  
What advantages does Kodo have over other persistence APIs?

Kodo is much less intrusive than other persistence APIs, in that your code does not need to be constantly "polluted" with additional code to do things like traversing relations. In addition, Kodo's bytecode enhancement allows performance advantages that cannot be matched by other reflection-based persistence architectures, such as declarative "fetch groups", automatic change detection, and transparent relation traversal.

- 1.3.8.  
Can Kodo be used in conjunction with other applications that operate against the same database?

Yes. Kodo does not require that it have exclusive read or write access to your database. The only restriction is that some optimistic lock strategies may need to be respected by other applications. For locking column considerations, see **Section 7.7, “Version Indicator” [229]**

- 1.3.9.  
Does Kodo provide extensions to the JDO API?

As well as providing complete support for the core JDO API, Kodo does provide API extensions for some advanced or JDB-specific operations. Many of Kodo's API extensions are under consideration for inclusion in the next major version of the JDO specification.

- 1.3.1  
0. How do I avoid vendor lock-in when using Kodo?

To avoid tying your application to any particular JDO vendor, you should avoid using any non-standard API extensions (i.e., avoid using any APIs that are not in the "javax.jdo" package). This will ensure that you can write your application in a way that will run in exactly the same way with any JDO compliant software.

- 1.3.1  
1. Is the application that I write in Kodo portable to other JDO vendors?

Yes, provided that the vendor's JDO implementation is truly compliant with the JDO specification. Since the JDO specification mandates that bytecode enhancement be done in a portable way, you have both source and binary portability of your JDO application.

- 1.3.1  
2. How does JDO interact with the data access/transfer pattern (DAO / DTO)?

While you can wrap JDO in the DAO pattern, most users find it easier to use JDO APIs directly. JDO provides much of DAO's functionality in a standardized and easy to use API set.

- 1.3.1  
3. How does Kodo affect the build process of my application?

The only change to your build process will typically be the addition of an "enhancement" phase.

- 1.3.1  
4. What is bytecode enhancement?

Bytecode enhancement is the mechanism by which JDO achieves persistence transparency. It involves running a tool on those classes that you have declared to be persistent. The class files will be changed internally to mediate access to all the fields that are marked as transparent. For more details, see **Section 4.1, "JDO Enhancer" [16]**.

- 1.3.1  
5. How can I debug enhanced persistent classes?

Kodo conforms to the mandate in the JDO specification that enhanced classes retain their line number tables. This means that lines in stack traces will match those of your original Java source file. Furthermore, enhanced classes can be used by debuggers in exactly the same way as unenhanced classes.

- 1.3.1  
6. Can I use JDO without having to enhance my classes?

JDO does not require bytecode enhancement to function, although not using enhancement involve writing all your persistent classes to implement the `javax.jdo.spi.PersistenceCapable` interface. As well as removing some of the transparency of JDO, it makes the process considerably more complex. For an example of using JDO without enhancement, see the `noenhancement` sample application (**Section 1.9, "Using Persistent Classes Without Enhancement" [370]**)

1.3.1

7. What is the differences between datastore caching and query caching?

Datastore caching caches data from the database. Basically, it caches your persistent objects. E.g. if you have a Person with id 100, and it's in the cache, if you try to reference Person with id 100, it won't have to go to the database to get the data.

The query cache, on the other hand, caches the results of a query. So, e.g., if you did a query for people whose "salary > 100" (and whatever other parameters, etc. that you want), and get back a Person- 100, Person-101, Person-102, then the next time you do that same query, you'll get back the same objects (unless you invalidate the query by changing objects, or manually).

So, an example of where these are different are if you did a query for "salary > 100" and then a query for "salary > 1000", the second query would be run against the database (it couldn't use the prior cached query), but all of the retrieved objects would be in the datastore cache, so the data for them wouldn't need to be retrieved.

1.3.1

8. How should I choose among the different ways to store my mapping information?

There are four common places that you can store mapping information in Kodo. Choosing the right one for your application can be daunting. In general, we recommend that people stick with the default (store mapping information in a separate .mapping file), as it keeps the JDO metadata files and your source code clean. Further, we recommend putting all your mapping information for a given package into a package.mapping file, rather than using separate mapping files for each class.

However, other factors come into play as well. If you use a number of different databases at the same time, you may want to use the db mapping factory, so that you can easily have different mappings for different databases. If you like to have a single source document per class, then maybe XDoclet tags are the way to go. See [Section 7.3, "Mapping Factory" \[214\]](#) for a detailed description of the different options available to you.

However, wherever you put them, remember that mapping information must be complete. That is, you cannot list partial information for just those fields whose column names you want to rename. This can mean that things can get a bit verbose when using XDoclet extensions.

1.3.1

9. When using the mappingtool to create a schema for me, why does Kodo create a column called NAME0 for a field called name?

When Kodo automatically generates a column name for a field, it ensures that the name fits within the limitations of your database. This means that a long field name might be truncated, and that fields whose names are common SQL keywords (such as name) will have a 0 appended to the column name.

If the auto-generated column names bother you, you can always manually edit your mapping information to control exactly what your schema should look like.

## 1.4. How do I ... ?

---

1.4.1.

I would like to have dynamic control over fetch groups at runtime. Is this possible?

Yes, using Kodo's custom fetch groups. To have complete control over the fields that are loaded when the persistent object is instantiated, you could declare each field as belonging to a different custom fetch group, and then at runtime set up the fields to be loaded for a specific operation. See [Section 14.5, "Fetch Groups" \[337\]](#)

- 1.4.2. I want to decouple a persistent instance from its `PersistenceManager` and `Transaction`. How can I do this?

There are a number of ways to decouple an instance from the `PersistenceManager`. The simplest is to just call `makeTransient()` on the object, which will decouple the object (but not related objects) from the `PersistenceManager`. If you want to decouple the object as well as all its relations, you can serialize the object and then deserialize it, but this will traverse the entire object graph, which could potentially draw down the entire contents of the database, with catastrophic memory and performance consequences. The third way is to use Kodo's attach and detach extensions, which allows an object and all those fields in the current fetch group for the object be detached (and then possibly serialized later). For information about detaching instances, see [Section 11.1, “Detach and Attach” \[297\]](#)

- 1.4.3. How do I directly access the JDBC Connection that Kodo is using?

Kodo provides a number of ways to access the underlying JDBC connection that the `PersistenceManager` is using. See [Section 4.9, “Runtime Access to JDBC Connections” \[179\]](#)

- 1.4.4. Can I see the SQL that Kodo is issuing to the database?

Yes. You can enable the SQL logging channel to see all the SQL statements that are sent from the database. You can also enable the JDBC channel to see most of the JDBC operations (such as commit and rollback operations) that are executed. For details on logging configuration, see [Chapter 3, Logging \[161\]](#)

- 1.4.5. Can Kodo use my existing logging framework instead of its own?

Yes. Kodo has built-in support for Log4J and the Apache Commons Logging frameworks. In turn, the Commons Logging framework can be configured to use JDK 1.4 `java.util.logging`. Additionally, it is possible to plug in your own logging implementation to override Kodo's default behavior. For details on logging configuration, see [Chapter 3, Logging \[?\]](#).

- 1.4.6. I would like to execute my queries in raw SQL rather than using JDOQL. Is this possible?

Yes. Kodo allows you to directly execute SQL statements and have the results returned as instances of your persistent classes. See [Section 13.4, “Direct SQL Execution” \[324\]](#) This can be useful for migrating your queries from a JDBC application to a JDO application.

- 1.4.7. How do I issue a query against a `Date` field?

You need to use a parameter to for the query. For details, see [Section 11.2, “JDOQL” \[56\]](#). An example of this is:

***Example 1.1. Issuing a query against a `Date` field***

```
Query query = myPersistenceManager.newQuery ("dateField < now");
query.declareParameters ("java.util.Date now");
Collection results = (Collection)query.execute (new Date ());
```

- 1.4.8. When is the object id available to be read when creating a new persistent instance?

When you first ask for the object id (via `JDOHelper.getObjectId` or `PersistenceManager.getObjectId`), or on flush -- whatever happens first. In fact, if the identity of your object depends on auto-increment columns, asking for the object id can cause an implicit flush to get the database-generated primary key value(s).

Once you have flushed or retrieved the id of an object, that id is permanent. If the object uses application identity, attempting to change any primary key fields will cause an exception.

- 1.4.9. How do I do query-by-example in Kodo?

You can provide a template object for Kodo to compare to by using a non-persistent parameter to a query. See [Example 11.8, “Query By Example” \[60\]](#) for details on how this works.

---

## 1.5. Common errors

- 1.5.1. When using application identity, what can cause strange behavior like objects not being found?

The most common cause of problems like these when using application identity is failure for the application identity to properly override the `equals()` and `hashCode()` methods as defined at [Section 4.5.2, “Application Identity” \[23\]](#).

- 1.5.2. What causes connection errors when returning persistent instances from a session bean?

A common cause of this problem is due to the EJB container serializing the instances that are being returned from the EJB. The serialization process happens at a point in the EJB lifecycle where the current status of the transaction is undefined. Since serialization may result in unloaded relations being traversed, Kodo will try to obtain a JDBC Connection to perform the traversal, and the application server may then disallow the connection access due to an invalid transaction status. The simplest solution to this is to either make the object to be returned transient, or return a detached instance, or manually perform the serialization before returning from the EJB method.

- 1.5.3. Why do my relations get lost after I commit?

The problem may be that you have defined an "inverse" extension to have two-sided relations, but you did not set both sides of the relation. Kodo does not perform any "magic" to keep relations consistent; the application must always ensure that the Java object model is consistent.

---

## 1.6. Productivity tools

- 1.6.1. Can Kodo be used with Apache Ant?

Apache Ant is a very popular build tool for Java projects. Kodo has full support for Ant by providing custom Ant tasks for

all of the Kodo development tasks. See [Section 15.2, “Apache Ant” \[346\]](#)

- 1.6.2.  
Does Kodo work my my IDE?

Kodo has plug-ins for many popular IDEs, such as JBuilder, IBM WSAD, Eclipse, and Netbeans. For those IDEs for which Kodo does not provide a plug-in, developers can use the Apache Ant integration to participate in the build process. Most IDEs have the capability of using Ant as their build tool. See [Chapter 15, \*Third Party Integration\* \[346\]](#)

- 1.6.3.  
Can I use XDoclet to generate my metadata?

XDoclet is a popular tool that allows developers to embed metadata in specially-formatted source code comments. The XDoclet distribution provides a JDO doclet that enables the generation of JDO metadata files. See [Section 15.3, “XDoclet” \[351\]](#)

## 1.7. Performance

---

- 1.7.1.  
Does Kodo use any caching?

Yes. Kodo has two levels of caching. The first is a per-PersistenceManager cache that is mandated by the JDO specification. Kodo also provides a level 2 cache that can be shared by multiple PersistenceManager instances and has the capability of synchronizing across a distributed system. See [Section 14.3, “Datastore Cache” \[329\]](#)

- 1.7.2.  
Is Kodo faster than JDBC?

A Kodo application will typically outperform a generic JDBC application, since Kodo is able to efficiently batch together like operations at commit time, eliminate redundant SQL, and perform sophisticated caching. Since Kodo runs atop a JDBC driver, it will never be able to communicate with the database any faster than the JDBC driver can. However, for any operations beyond the most simplistic JDBC program, Kodo will dramatically increase the performance of any application beyond what is realistically possible using raw JDBC.

- 1.7.3.  
Is Kodo faster than EJBs?

It is only meaningful to compare the performance of Kodo with that of CMP Entity Beans. While the performance of CMP beans varies depending on the application server being used, they will almost always be slower than using JDO persistent objects, since JDO does not incur the heavy method invocation overhead that penalizes all EJBs.

- 1.7.4.  
How can I speed up my Kodo application?

Kodo provides a wide variety of configuration options and API extensions to fine tune your applications performance. For an overview of common optimization techniques, see [Chapter 16, \*Optimization Techniques\* \[362\]](#)

- 1.7.5.  
How can I profile the performance of my application?

Kodo provides a sophisticated set of management tools to analyze the behavior and performance of your application. This is described in the section on management. Also, Kodo can be used in all popular profiling applications such as OptimizeIt or JProbe.

1.7.6.

Can I customize the fields that Kodo loads when an object is first instantiated from the database?

Yes. JDO defines the notion of a "default-fetch-group", which enables the application to specify the fields that will be loaded whenever a persistent instance is instantiated from the database. Adding to that, Kodo provides extensions that allow the definition of custom fetch groups, which enable the application to dynamically specify which fields to instantiate eagerly. See [Section 10.5, "Fetch Configuration" \[290\]](#)

1.7.7.

When I traverse a relation, Kodo issues a separate statement to the database. How can I ensure that the relation is always loaded immediately?

Relations, like any other fields, can be added to the default fetch group, which will cause Kodo to attempt to efficiently traverse the relation when the owning persistent instance is first instantiated from the database.

1.7.8.

Does Kodo utilize connection pooling?

Kodo provides its own built-in connection pooling framework. Additionally, Kodo can be integrated with any third-party connection pool that implements the `javax.sql.DataSource` interface (including the connection pools that are provided with all known application servers). See [Section 4.1, "Using the Kodo JDO DataSource" \[166\]](#) and [Section 4.2, "Using a Third-Party DataSource" \[168\]](#)

## 1.8. Scalability

---

1.8.1.

Does Kodo support load balancing?

Yes, you can easily use Kodo in conjunction with a server farm or cluster, with a load balancer in front of the farm. Kodo provides support for synchronizing its caches across JVMs, so you can use Kodo's data cache when in a clustered environment as well.

1.8.2.

I have to process millions of records, but my servers do not have enough memory to hold all of the records in memory at the same time. Can Kodo handle doing that?

Yes. By default, Kodo does not hold hard references to objects read from the database, so, provided that you do not hold hard references to more objects than you can fit into memory, you will not run out of memory when iterating through large data sets.

## 1.9. Application servers

---

1.9.1.

Can Kodo be run inside an application server?

Kodo can be used in any J2EE compliant application server. Kodo integrates with managed environments (such as application servers) in a variety of ways, from synchronization with container managed transactions to support for accessing Data-Sources from JNDI.

1.9.2.

Which application servers does Kodo support?

Kodo supports any J2EE compliant application server. For ease of configuration and deployment, Kodo recommends (but does not require) using an application server that supports the Java Connector Architecture (JCA). Kodo has been tested with most popular application servers such as JBoss, BEA Weblogic, IBM Websphere, SunONE, Macromedia JRun, and Borland Enterprise Server. See **Chapter 4, *J2EE Tutorial* [103]**.

1.9.3.

How can I integrate Kodo's transactions with the application server's transaction?

Yes. The JDO specification defines that a JDO compliant implementation will integrate its own transaction with the current global transaction of a managed environment.

1.9.4.

Can I use JDO to implement my entity EJBs?

It is possible to use Kodo to implement bean managed persistence (BMP) entity EJBs. However, doing so will introduce the performance penalties incurred by entity beans. The recommended pattern is to use session beans to perform fine-grained persistence operations with Kodo. To get started with using Kodo with EJBs, see **Chapter 4, *J2EE Tutorial* [103]**.

## 1.10. Locking

---

1.10.

1. What types of locking does Kodo provide?

Kodo provides support for both datastore locking (which is required by the JDO specification) and optimistic locking (which is optional in the JDO specification). Datastore locking is implemented using pessimistic database operations that acquire locks on all objects that are retrieved in a JDO transaction. Optimistic locking, on the other hand, will verify that no other transaction has modified any of the objects at commit time, and throw an exception if the object has been changed. See **Section 9.1, “Transaction Types” [50]**.

Advanced users can get more fine-grained control over locking using Kodo's object locking APIs. See **Section 10.8, “Object Locking” [292]**

1.10.

2. Which kind of locking should I use for my application?

The answer depends greatly on the type of application you are developing. The advantage of using pessimistic locking is that the application does not need to worry about any locking violations at commit time, but at the cost of potentially introducing serious locking contention into your application. The advantage of using optimistic locking is that it can be much faster and makes more efficient use of database connections, since it does not necessarily need to hold open a single connection for the duration of a transaction. The primary disadvantage of optimistic locking is that the application must catch lock violation exceptions at commit time and take the appropriate actions upon failure (such as re-trying the operation, or telling the user a violation has occurred). Advanced users who want something between these extremes can use Kodo's fine-grained object locking APIs, detailed in **Section 10.8, “Object Locking” [292]**

1.10.

3. Why do I get optimistic exceptions even though I am using pessimistic transactions?

If your objects have an optimistic version indicator, then Kodo will still perform optimistic locking logic even if you are using pessimistic transactions. This is so that you can mix optimistic and pessimistic transactions over the life cycle of your application.

## 1.11. Transactions

---

1.11.

1. How does Kodo use transactions?

The JDO specification defines a `javax.jdo.Transaction` interface that is similar to a JTA transaction. The JDO application will begin a transaction before making changes to objects, and then commit (or rollback) the transaction once the operation is complete. JDO does not mandate any specific transaction demarcation boundaries; the application is free to begin and end transactions in the way that the developer deems suitable. See [Chapter 9, \*Transaction\* \[50\]](#).

1.11.

2. Does a JDO transaction translate directly to a database transaction?

Not necessarily. For optimistic JDO transactions, Kodo will typically only begin a database transaction when the JDO transaction is committed, or if changes are manually flushed to the database. For pessimistic transactions, Kodo will begin a database transaction only once the first objects are acquired and locked.

---

## **Part V. Kodo JDO Reference Guide**

---

---

# Table of Contents

1. Introduction .....	138
1.1. Intended Audience .....	138
2. Configuration .....	139
2.1. Introduction .....	139
2.2. Runtime Configuration .....	139
2.3. Command Line Configuration .....	139
2.3.1. Code Formatting .....	140
2.4. Plugin Configuration .....	140
2.5. JDO Standard Properties .....	141
2.5.1. javax.jdo.PersistenceManagerFactoryClass .....	141
2.5.2. javax.jdo.option.ConnectionDriverName .....	142
2.5.3. javax.jdo.option.ConnectionFactoryName .....	142
2.5.4. javax.jdo.option.ConnectionFactory2Name .....	142
2.5.5. javax.jdo.option.ConnectionPassword .....	142
2.5.6. javax.jdo.option.ConnectionURL .....	143
2.5.7. javax.jdo.option.ConnectionUserName .....	143
2.5.8. javax.jdo.option.IgnoreCache .....	143
2.5.9. javax.jdo.option.Multithreaded .....	143
2.5.10. javax.jdo.option.NontransactionalRead .....	143
2.5.11. javax.jdo.option.NontransactionalWrite .....	144
2.5.12. javax.jdo.option.Optimistic .....	144
2.5.13. javax.jdo.option.RestoreValues .....	144
2.5.14. javax.jdo.option.RetainValues .....	144
2.6. Kodo JDO Properties .....	145
2.6.1. kodo.AggregateListeners .....	145
2.6.2. kodo.ClassResolver .....	145
2.6.3. kodo.ConnectionProperties .....	145
2.6.4. kodo.ConnectionFactoryProperties .....	145
2.6.5. kodo.Connection2DriverName .....	146
2.6.6. kodo.Connection2Password .....	146
2.6.7. kodo.Connection2URL .....	146
2.6.8. kodo.Connection2UserName .....	146
2.6.9. kodo.Connection2Properties .....	147
2.6.10. kodo.ConnectionFactory2Properties .....	147
2.6.11. kodo.ConnectionRetainMode .....	147
2.6.12. kodo.CopyObjectIds .....	147
2.6.13. kodo.DataCache .....	148
2.6.14. kodo.DataCacheTimeout .....	148
2.6.15. kodo.DynamicDataStructs .....	148
2.6.16. kodo.EagerFetchMode .....	148
2.6.17. kodo.FetchBatchSize .....	149
2.6.18. kodo.FetchGroups .....	149
2.6.19. kodo.FilterListeners .....	149
2.6.20. kodo.FlushBeforeQueries .....	149
2.6.21. kodo.InverseManager .....	149
2.6.22. kodo.LicenseKey .....	150
2.6.23. kodo.LockManager .....	150
2.6.24. kodo.LockTimeout .....	150
2.6.25. kodo.Log .....	150
2.6.26. kodo.ManagedRuntime .....	151
2.6.27. kodo.ManagementConfiguration .....	151
2.6.28. kodo.MetadataLoader .....	151
2.6.29. kodo.ObjectLookupMode .....	151

2.6.30. kodo.OrphanedKeyAction .....	152
2.6.31. kodo.PersistenceManagerImpl .....	152
2.6.32. kodo.PersistenceManagerServer .....	152
2.6.33. kodo.PersistentClasses .....	152
2.6.34. kodo.ProxyManager .....	153
2.6.35. kodo.QueryCache .....	153
2.6.36. kodo.QueryCompilationCache .....	153
2.6.37. kodo.ReadLockLevel .....	153
2.6.38. kodo.RemoteCommitProvider .....	154
2.6.39. kodo.RestoreMutableValues .....	154
2.6.40. kodo.RetainValuesInOptimistic .....	154
2.6.41. kodo.RetryClassRegistration .....	154
2.6.42. kodo.SubclassFetchMode .....	155
2.6.43. kodo.TransactionMode .....	155
2.6.44. kodo.WriteLockLevel .....	155
2.6.45. kodo.jdbc.ClassIndicator .....	155
2.6.46. kodo.jdbc.ConnectionDecorators .....	156
2.6.47. kodo.jdbc.DataSourceMode .....	156
2.6.48. kodo.jdbc.DBDictionary .....	156
2.6.49. kodo.jdbc.FetchDirection .....	156
2.6.50. kodo.jdbc.ForeignKeyConstraints .....	157
2.6.51. kodo.jdbc.JDBCListeners .....	157
2.6.52. kodo.jdbc.LRSSize .....	157
2.6.53. kodo.jdbc.MappingFactory .....	157
2.6.54. kodo.jdbc.ResultSetType .....	158
2.6.55. kodo.jdbc.SchemaFactory .....	158
2.6.56. kodo.jdbc.Schemas .....	158
2.6.57. kodo.jdbc.SequenceFactory .....	158
2.6.58. kodo.jdbc.SubclassMapping .....	159
2.6.59. kodo.jdbc.SynchronizeMappings .....	159
2.6.60. kodo.jdbc.TransactionIsolation .....	159
2.6.61. kodo.jdbc.UpdateManager .....	159
2.6.62. kodo.jdbc.VersionIndicator .....	159
3. Logging .....	161
3.1. Logging Channels .....	161
3.2. Kodo Logging .....	162
3.3. Disabling Logging .....	163
3.4. Log4J .....	163
3.5. Apache Commons Logging .....	164
3.5.1. JDK 1.4 java.util.logging .....	164
3.6. Custom Log .....	165
4. JDBC .....	166
4.1. Using the Kodo JDO DataSource .....	166
4.2. Using a Third-Party DataSource .....	168
4.2.1. Enlisted Data Sources .....	169
4.3. Database Support .....	169
4.3.1. MySQLDictionary parameters .....	175
4.3.2. OracleDictionary parameters .....	175
4.4. Configuring the DBDictionary .....	176
4.5. Accessing Multiple Databases .....	176
4.6. Setting the Transaction Isolation .....	176
4.7. Setting the SQL Join Syntax .....	177
4.8. Configuring the Use of JDBC Connections .....	177
4.9. Runtime Access to JDBC Connections .....	179
4.10. Large Result Sets .....	180
4.11. SQL Statement Ordering & Foreign Keys .....	182
5. Persistent Classes .....	184
5.1. Restrictions on Persistent Classes .....	184
5.2. Object Identity .....	184

5.2.1. Datastore Identity .....	184
5.2.2. Application Identity .....	184
5.2.3. Single Field Identity .....	185
5.2.4. Primary Key Generation .....	185
5.2.4.1. Sequence Factory .....	185
5.2.4.2. Auto-Increment .....	187
5.2.4.3. Sequence-Assigned .....	188
5.3. Managed Inverses .....	188
5.4. Mutable Second Class Object Fields .....	189
5.4.1. Restoring Mutable Fields .....	189
5.4.2. Typing and Ordering .....	189
5.4.3. Proxies .....	190
5.4.3.1. Smart Proxies .....	190
5.4.3.2. Large Result Set Proxies .....	190
5.4.3.3. Custom Proxies .....	191
5.5. Enhancement .....	192
5.6. Auto-Generating Classes from a Schema .....	193
5.6.1. Customizing Reverse Mapping .....	195
5.7. Persistent Class List .....	196
6. Metadata .....	197
6.1. Generating Default JDO Metadata .....	197
6.2. JDO Metadata Extensions .....	197
6.2.1. Relation Extensions .....	198
6.2.1.1. inverse-owner .....	198
6.2.1.2. inverse-logical .....	198
6.2.1.3. dependent .....	198
6.2.1.4. element-dependent .....	199
6.2.1.5. value-dependent .....	199
6.2.1.6. key-dependent .....	199
6.2.1.7. type .....	199
6.2.1.8. element-type .....	199
6.2.1.9. value-type .....	200
6.2.1.10. key-type .....	200
6.2.1.11. lrs .....	200
6.2.1.12. Example .....	200
6.2.2. Schema Extensions .....	200
6.2.2.1. jdbc-size .....	200
6.2.2.2. jdbc-element-size .....	200
6.2.2.3. jdbc-value-size .....	201
6.2.2.4. jdbc-key-size .....	201
6.2.2.5. jdbc-type .....	201
6.2.2.6. jdbc-sql-type .....	201
6.2.2.7. jdbc-indexed .....	201
6.2.2.8. jdbc-element-indexed .....	201
6.2.2.9. jdbc-value-indexed .....	201
6.2.2.10. jdbc-key-indexed .....	201
6.2.2.11. jdbc-ref-indexed .....	201
6.2.2.12. jdbc-version-ind-indexed .....	202
6.2.2.13. jdbc-class-ind-indexed .....	202
6.2.2.14. jdbc-delete-action .....	202
6.2.2.15. jdbc-element-delete-action .....	202
6.2.2.16. jdbc-value-delete-action .....	203
6.2.2.17. jdbc-key-delete-action .....	203
6.2.2.18. jdbc-ref-delete-action .....	203
6.2.2.19. Example .....	203
6.2.3. Object-Relational Mapping Extensions .....	203
6.2.3.1. jdbc-class-map-name .....	203
6.2.3.2. jdbc-version-ind-name .....	204
6.2.3.3. jdbc-class-ind-name .....	204

6.2.3.4. jdbc-field-map-name .....	204
6.2.3.5. jdbc-field-mappings .....	204
6.2.3.6. jdbc-ordered .....	204
6.2.3.7. jdbc-container-meta .....	204
6.2.3.8. jdbc-null-ind .....	205
6.2.3.9. externalizer .....	205
6.2.3.10. factory .....	205
6.2.3.11. external-values .....	205
6.2.3.12. jdbc-class-ind-value .....	205
6.2.3.13. Example .....	205
6.2.4. Miscellaneous Extensions .....	206
6.2.4.1. detachable .....	206
6.2.4.2. detached-objectid-field .....	206
6.2.4.3. detached-state-field .....	206
6.2.4.4. fetch-group .....	206
6.2.4.5. lock-group .....	206
6.2.4.6. lock-groups .....	206
6.2.4.7. data-cache .....	206
6.2.4.8. data-cache-timeout .....	206
6.2.4.9. sequence-assigned .....	206
6.2.4.10. subclass-fetch-mode .....	207
6.2.4.11. eager-fetch-mode .....	207
6.2.4.12. jdbc-sequence-factory .....	207
6.2.4.13. jdbc-sequence-name .....	207
6.2.4.14. jdbc-auto-increment .....	207
6.2.4.15. Example .....	207
7. Object-Relational Mapping .....	209
7.1. Mapping Tool .....	209
7.1.1. Using the Mapping Tool .....	210
7.1.2. Generating DDL SQL .....	212
7.2. Automatic Runtime Mapping .....	213
7.3. Mapping Factory .....	214
7.3.1. Importing and Exporting Mapping Data .....	215
7.4. Mapping File XML Format .....	215
7.5. Mapping Notes .....	217
7.5.1. Join Attributes .....	218
7.5.2. Non-Standard Joins .....	218
7.6. Class Mapping .....	219
7.6.1. Base Mapping .....	220
7.6.2. Flat Inheritance Mapping .....	220
7.6.2.1. Advantages of using Flat Inheritance Mapping .....	222
7.6.2.2. Disadvantages of using Flat Inheritance Mapping .....	222
7.6.3. Vertical Inheritance Mapping .....	222
7.6.3.1. Advantages of using Vertical Inheritance Mapping .....	223
7.6.3.2. Disadvantages of using Vertical Inheritance Mapping .....	224
7.6.3.3. Vertical Select Modes .....	224
7.6.4. Horizontal Inheritance Mapping .....	224
7.6.4.1. Special considerations when using Horizontal Inheritance Mapping .....	228
7.6.4.2. Advantages of using Horizontal Inheritance Mapping .....	228
7.6.4.3. Disadvantages of using Horizontal Inheritance Mapping .....	228
7.6.5. Custom Class Mapping .....	228
7.7. Version Indicator .....	229
7.7.1. Version Number Indicator .....	229
7.7.2. Version Date Indicator .....	230
7.7.3. State Image Indicator .....	231
7.7.4. Custom Version Indicator .....	232
7.8. Class Indicator .....	232
7.8.1. In-Class-Name Indicator .....	232
7.8.2. Metadata Value Indicator .....	233

7.8.3. Subclass-Join Indicator .....	235
7.8.4. Custom Class Indicator .....	236
7.9. Field Mapping .....	236
7.9.1. Value Mapping .....	236
7.9.2. Blob Mapping .....	238
7.9.3. Clob Mapping .....	240
7.9.4. Byte Array Mapping .....	241
7.9.5. One-to-One Mapping .....	242
7.9.6. PC One-to-One Mapping .....	246
7.9.7. Embedded One-to-One Mapping .....	248
7.9.8. Enumeration Mapping .....	250
7.9.9. Collection Mapping .....	251
7.9.10. Many-to-Many Mapping .....	253
7.9.11. One-to-Many Mapping .....	256
7.9.12. PC Collection Mapping .....	259
7.9.13. Map Mapping .....	260
7.9.14. N-to-Many Map Mapping .....	261
7.9.15. Many-to-N Map Mapping .....	263
7.9.16. Many-to-Many Map Mapping .....	264
7.9.17. PC Map Mapping .....	266
7.9.18. N-to-PC Map Mapping .....	267
7.9.19. PC-to-N Map Mapping .....	269
7.9.20. PC-to-Many Map Mapping .....	270
7.9.21. Many-to-PC Map Mapping .....	272
7.9.22. Custom Field Mapping .....	273
7.9.23. Externalization .....	273
7.9.24. External Values .....	276
8. Schema Information .....	278
8.1. Schema Reflection .....	278
8.1.1. Schemas List .....	278
8.1.2. Schema Factory .....	278
8.1.3. Schema Generator .....	279
8.2. Schema Tool .....	280
8.3. XML Schema Format .....	282
8.4. The SQLLine Utility .....	284
9. Runtime Deployment .....	286
9.1. JDOHelper .....	286
9.2. KodoHelper .....	286
9.3. J2EE Deployment .....	287
10. JDO Runtime Extensions .....	288
10.1. KodoPersistenceManagerFactory .....	288
10.2. KodoPersistenceManager .....	288
10.2.1. JDO Transaction Events .....	288
10.2.2. JDO 2 Preview Methods .....	288
10.2.3. Lifecycle Events .....	289
10.2.4. PersistenceManager Extension .....	289
10.3. KodoExtent .....	290
10.4. KodoQuery .....	290
10.5. Fetch Configuration .....	290
10.6. KodoHelper .....	290
10.7. Query Extensions .....	290
10.7.1. JDOQL Extensions .....	291
10.7.1.1. Included JDOQL Extensions .....	291
10.7.1.2. Developing Custom JDOQL Extensions .....	292
10.7.1.3. Configuring JDOQL Extensions .....	292
10.7.2. Aggregate Extensions .....	292
10.7.2.1. Configuring Query Aggregates .....	292
10.8. Object Locking .....	292
10.8.1. Configuring Default Locking .....	293

10.8.2. Configuring Lock Levels at Runtime .....	293
10.8.3. Object Locking APIs .....	293
10.8.4. Lock Manager .....	294
10.8.5. Rules for Locking Behavior .....	295
10.8.6. Known Issues and Limitations .....	295
10.9. Orphaned Keys .....	296
11. Remote and Offline JDO .....	297
11.1. Detach and Attach .....	297
11.1.1. Declaring Detachability .....	297
11.1.2. Detach and Attach Behavior .....	298
11.1.3. Defining the Detached Object Graph .....	299
11.1.4. Detach and Attach Callbacks .....	301
11.1.5. Automatic Detachment .....	301
11.1.5.1. Detach on Close .....	301
11.1.5.2. Detach on Serialize .....	302
11.2. Remote Persistence Managers .....	303
11.2.1. Standalone Persistence Manager Server .....	304
11.2.2. HTTP Persistence Manager Server .....	305
11.2.3. Client Persistence Managers .....	305
11.2.4. Data Compression and Filtering .....	307
11.2.5. Remote Persistence Manager Deployment .....	308
12. Management and Monitoring .....	309
12.1. Configuration .....	309
12.1.1. Optional Parameters in Management Group .....	311
12.1.2. Optional Parameters in Remote Group .....	311
12.1.3. Optional Parameters in JSR 160 Group .....	312
12.1.4. Optional Parameters in WebLogic 8.1 Group .....	312
12.1.5. Configuring Logging for Management / Monitoring .....	313
12.2. Kodo Management Console .....	313
12.2.1. Remote Connection .....	313
12.2.1.1. Connecting to Kodo under WebLogic 8.1 .....	314
12.2.1.2. Connecting to Kodo under JBoss 3.2 .....	314
12.2.1.3. Connecting to Kodo under JBoss 4 .....	315
12.2.2. Using the Kodo Management Console .....	315
12.2.2.1. JMX Explorer .....	316
12.2.2.1.1. Executing Operations .....	316
12.2.2.1.2. Listening to Notifications .....	317
12.2.2.2. MBean Panel .....	317
12.2.2.2.1. Notifications / Statistics .....	317
12.2.2.2.2. Setting Attributes .....	317
12.3. Accessing the MBeanServer from Code .....	317
12.4. MBeans .....	318
12.4.1. Log MBean .....	318
12.4.2. Kodo Pooling DataSource MBean .....	318
12.4.3. PreparedStatement Cache MBean .....	318
12.4.4. Query Cache MBean .....	318
12.4.5. Datastore Cache MBean .....	318
12.4.6. TimeWatch MBean .....	318
12.4.7. Runtime MBean .....	319
12.4.8. Profiling MBean .....	319
13. Enterprise Edition .....	321
13.1. Integrating with the Transaction Manager .....	321
13.2. XA Transactions .....	321
13.2.1. Requirements for Using Kodo with XA Transactions .....	322
13.2.2. Configuring Kodo to Utilize XA Transactions .....	322
13.3. JDOQL Subqueries .....	323
13.3.1. Subquery Parameters, Variables, and Imports .....	324
13.4. Direct SQL Execution .....	324
13.5. MethodQL .....	324

13.6. Remote PersistenceManagers .....	325
13.7. Custom Class Mappings .....	325
13.8. Non-relational Database Access .....	325
14. Performance Pack .....	326
14.1. SQL Batching .....	326
14.2. Eager Fetching .....	326
14.2.1. Configuring Eager Fetching .....	327
14.2.2. Eager Fetching Considerations .....	328
14.3. Datastore Cache .....	329
14.3.1. Overview of Kodo JDO Datastore Caching .....	329
14.3.2. Kodo JDO Cache Usage .....	329
14.3.3. Query Caching .....	332
14.3.4. DataCache Integrations .....	334
14.3.5. Cache Extension .....	334
14.3.6. Important Notes .....	334
14.3.7. Known Issues and Limitations .....	335
14.4. Remote Event Notification Framework .....	335
14.4.1. Remote Commit Provider Configuration .....	336
14.4.2. Customization .....	337
14.5. Fetch Groups .....	337
14.5.1. Normal Default Fetch Group Behavior .....	337
14.5.2. Kodo JDO Fetch Group Behavior .....	338
14.5.3. Custom Fetch Group Configuration .....	339
14.5.4. Per-field Fetch Configuration .....	339
14.6. Lock Groups .....	340
14.6.1. Lock Groups and Subclasses .....	341
14.6.2. Lock Group Mapping .....	342
14.7. Profiling .....	342
14.7.1. Profiling in an embedded GUI .....	343
14.7.2. Dumping profiling data to disk from a batch process .....	345
14.7.3. Controlling how the profiler obtains context information .....	345
15. Third Party Integration .....	346
15.1. Overview of Third Party Integration features in Kodo .....	346
15.2. Apache Ant .....	346
15.2.1. Common Ant Configuration Options .....	346
15.2.2. JDO Enhancer Ant Task .....	348
15.2.3. Application Identity Tool Ant Task .....	348
15.2.4. JDO Metadata Tool Ant Task .....	349
15.2.5. Mapping Tool Ant Task .....	349
15.2.6. Reverse Mapping Tool Ant Task .....	349
15.2.7. Schema Tool Ant Task .....	350
15.2.8. Schema Generator Ant Task .....	350
15.3. XDoclet .....	351
15.4. Borland JBuilder .....	355
15.4.1. Installing Kodo Into JBuilder .....	355
15.4.2. Kodo Configuration from JBuilder .....	356
15.4.3. Creating and building JDO projects in JBuilder .....	356
15.4.4. Editing JDO Metadata from JBuilder .....	356
15.4.5. Editing Mapping Info from JBuilder .....	356
15.4.6. JBuilder Project Sample .....	357
15.5. Sun ONE Studio / NetBeans IDE .....	357
15.5.1. Before Installing Kodo into the IDE .....	357
15.5.2. Installing Kodo into the IDE .....	358
15.5.3. Configuring the Kodo Module .....	358
15.5.4. Kodo Template Wizards .....	358
15.5.5. JDO DataObject .....	359
15.5.6. Mapping DataObject .....	359
15.5.7. Kodo Integration into the Build Process .....	359
15.5.8. SunONE / NetBeans Sample .....	359

15.6. Eclipse / WebSphere Studio Integration .....	360
15.6.1. Installing the Kodo Eclipse Plugin .....	360
15.6.2. Configuring the Plugin .....	361
15.6.3. Using Kodo in Eclipse IDEs .....	361
15.6.4. Eclipse Sample .....	361
16. Optimization Techniques .....	362

---

# Chapter 1. Introduction

Kodo JDO is a JDBC-based implementation of the JDO 1.0.2 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. We also **strongly** recommend taking Kodo JDO's hands-on **tutorials** to get comfortable with Kodo JDO basics.

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

## 2.1. Introduction

---

This chapter describes the Kodo JDO configuration frameworks and how to configure Kodo JDO for your data store and runtime environment. It concludes with descriptions of all the configuration properties recognized by Kodo JDO. You may want to browse these properties now, but it is not necessary. Most of them will be referenced later in the documentation as we explain the various features they apply to.

## 2.2. Runtime Configuration

---

In JDO, `PersistenceManagerFactory`s are usually obtained through the `JDOHelper.getPersistenceManagerFactory(Properties)` method. Kodo JDO's runtime settings can be entirely specified via the properties passed to this method. Kodo JDO also includes a comprehensive system of property defaults and overrides:

- All properties default 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 name or location of the above resource by specifying the correct resource path in the `kodo.properties` System property.
- You can override any default value defined in the `kodo.properties` resource by setting the System property of the same name to the desired value.
- The values in the `Properties` object passed to `JDOHelper.getPersistenceManagerFactory` at runtime override the default values in the `kodo.properties` resource and any System property settings.
- All Kodo JDO command-line tools accept flags that allow you to specify the properties file to use and to override certain properties. [Section 2.3, “Command Line Configuration” \[139\]](#) describes these flags.

### Note

Internally, the Kodo JDO runtime environment and development tools manipulate property settings through SolarMetric's general **Configuration** interface, and in particular its **JDOConfiguration** and **JDBCCConfiguration** subclasses. For advanced customization, Kodo JDO's `PersistenceManagerFactory` implementation and its development tools allow you to access these interfaces directly. See the **Javadoc** for details.

## 2.3. Command Line Configuration

---

Kodo JDO development tools share the same set of property defaults and overrides as the runtime system. They also allow you to specify property values on the command line:

- `-properties/-p <properties file or resource>` : Use the `-properties` flag, or its shorter `-p` form, to specify a properties file to use in place of `kodo.properties`. The flag value can be the path to a file, or the resource name of a file somewhere in the CLASSPATH.
- `-<property name> <property value>` : Any configuration property that can be specified in a properties file can be overridden with a command line flag. The flag name is always the last token of the corresponding property name, with the first letter in either upper or lower case. For example, to override the `JDO javax.jdo.option.ConnectionUserName` property, you could pass the `-connectionUserName <value>`

flag to any tool. Values set this way override both the values in the properties file and values set via System properties.

## 2.3.1. Code Formatting

---

Some Kodo JDO development tools generate Java code. These tools share a common set of command-line flags for formatting their 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. Defaults to `false`.
- `-codeFormat./-cf.scoreBeforeFieldName <true/t | false/f>`: Whether to prefix an underscore to names of private member variables. Defaults to `false`.
- `-codeFormat./-cf.linesBetweenSections <lines>` : The number of lines to skip between sections of code. Defaults to 2.

### *Example 2.1. Code Formatting with the Application Id Tool*

```
appidtool -p dev.properties -cf.spaceBeforeParen true *.jdo
```

## 2.4. Plugin Configuration

---

Because Kodo JDO is a highly customizable environment, many configuration properties relate to the creation and configuration of system plugins. Plugin properties have a constructor-like syntax that allows you to specify both what class to use for the plugin and how to configure the bean properties of the instantiated plugin instance. The easiest way to describe the plugin syntax is by example:

Kodo has a pluggable L2 caching mechanism that is controlled by the `kodo.DataCache` configuration property. Suppose that you have created a new class, `com.xyz.MyDataCache`, that you want Kodo to use for caching. You've made instances of `MyDataCache` configurable via two methods, `setCacheSize(int size)` and `setRemoteHost(String host)`. The sample properties file line below shows how you would tell Kodo to use an instance of your custom plugin with a max size of 1000 and a remote host of `cacheserver`.

```
kodo.DataCache: com.xyz.MyDataCache(CacheSize=1000, RemoteHost="cacheserver")
```

As you can see, plugin properties take a class name, followed by a comma-separated list of values for the plugin's bean properties in parentheses. Kodo will match each named property to a "setter" method in the instantiated plugin instance, and invoke the method with the given value (after converting the value to the right type, of course). The first letter of the property names can be in either upper or lower case. The following would also have been valid:

```
kodo.DataCache: com.xyz.MyDataCache(cacheSize=1000, remoteHost="cacheserver")
```

If you do not need to pass any property settings to a plugin, you can just name the class to use:

```
kodo.DataCache: com.xyz.MyDataCache
```

Similarly, if the plugin has a default class that you do not want to change, you can simply specify a list of property settings, without a class name. For example, Kodo's query cache companion to the data cache has a default implementation suitable to most users, but you still might want to change the query cache's size (it has a `CacheSize` property for this purpose):

```
kodo.QueryCache: CacheSize=1000
```

Finally, many of Kodo's built-in options for plugins have short alias names that you can use in place of the full class name. The data cache property, for example, has an available alias of `true` for the standard cache implementation:

```
kodo.DataCache: true
```

The standard cache implementation class also has a `CacheSize` property, so to both use the standard implementation and configure the size:

```
kodo.DataCache: true(CacheSize=1000)
```

The remainder of this chapter reviews the set of configuration properties Kodo JDO recognizes.

## 2.5. JDO Standard Properties

---

JDO recognizes many standard runtime properties, all of which Kodo JDO supports (these properties are also covered in [Section 7.2, “PersistenceManagerFactory Properties”](#) [37] of the JDO Overview).

### 2.5.1. javax.jdo.PersistenceManagerFactoryClass

---

**Property name:** `javax.jdo.PersistenceManagerFactoryClass`

**Configuration API:** `kodo.conf.JDOConfiguration.getPersistenceManagerFactoryClass`

**Resource adaptor config-property:** `PersistenceManagerFactoryClass`

**Default:** -

**Description:** The name of the concrete implementation of the `javax.jdo.PersistenceManagerFactory` that `javax.jdo.JDOHelper.getPersistenceManagerFactory` should create. For Kodo JDO, this should be `kodo.jdbc.runtime.JDBCPersistenceManagerFactory` or a custom extension of this type.

## 2.5.2. javax.jdo.option.ConnectionDriverName

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionDriverName`

**Resource adaptor config-property:** `ConnectionDriverName`

**Default:** -

**Description:** The full class name of either the JDBC `java.sql.Driver`, or a `javax.sql.DataSource` implementation to use to connect to the database. See [Chapter 4, JDBC \[166\]](#) for details.

## 2.5.3. javax.jdo.option.ConnectionFactoryName

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionFactoryName`

**Resource adaptor config-property:** `ConnectionFactoryName`

**Default:** -

**Description:** The JNDI location of a `javax.sql.DataSource` to use to connect to the database. See [Chapter 4, JDBC \[166\]](#) for details.

## 2.5.4. javax.jdo.option.ConnectionFactory2Name

---

**Property name:** `kodo.ConnectionFactory2Name`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionFactory2Name`

**Resource adaptor config-property:** `ConnectionFactory2Name`

**Default:** -

**Description:** The JNDI location of a non-XA `javax.sql.DataSource` to use to connect to the database. See [Section 13.2, “XA Transactions” \[321\]](#) for details.

## 2.5.5. javax.jdo.option.ConnectionPassword

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionPassword`

**Resource adaptor config-property:** `ConnectionPassword`

**Default:** -

**Description:** The password for the user specified in the `ConnectionUserName` property. See [Chapter 4, JDBC \[166\]](#) for details.

---

## 2.5.6. javax.jdo.option.ConnectionURL

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionURL`

**Resource adaptor config-property:** `ConnectionURL`

**Default:** -

**Description:** The JDBC URL for the database. See [Chapter 4, JDBC \[166\]](#) for details.

## 2.5.7. javax.jdo.option.ConnectionUserName

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionUserName`

**Resource adaptor config-property:** `ConnectionUserName`

**Default:** -

**Description:** The user name to use when connecting to the database. See the [Chapter 4, JDBC \[166\]](#) for details.

## 2.5.8. javax.jdo.option.IgnoreCache

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getIgnoreCache`

**Resource adaptor config-property:** `IgnoreCache`

**Default:** `true`

**Description:** Whether to consider modifications to persistent objects made in the current transaction when evaluating queries. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) of the JDO Overview for details.

## 2.5.9. javax.jdo.option.Multithreaded

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getMultithreaded`

**Resource adaptor config-property:** `Multithreaded`

**Default:** `false`

**Description:** Whether persistent instances and JDO components will be accessed by multiple threads at once. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

## 2.5.10. javax.jdo.option.NontransactionalRead

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getNontransactionalRead`

**Resource adaptor config-property:** `NontransactionalRead`

**Default:** `true`

**Description:** Whether the JDO runtime will allow you to read data outside of a transaction. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

---

## 2.5.11. `javax.jdo.option.NontransactionalWrite`

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getNontransactionalWrite`

**Resource adaptor config-property:** `NontransactionalWrite`

**Default:** `false`

**Description:** Whether you can modify persistent fields outside of a transaction. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

---

## 2.5.12. `javax.jdo.option.Optimistic`

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getOptimistic`

**Resource adaptor config-property:** `Optimistic`

**Default:** `true`

**Description:** Selects between optimistic and pessimistic (data store) transactional modes. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

---

## 2.5.13. `javax.jdo.option.RestoreValues`

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getRestoreValues`

**Resource adaptor config-property:** `RestoreValues`

**Default:** `true`

**Description:** Whether to restore managed fields to their pre-transaction values when a rollback occurs. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

---

## 2.5.14. `javax.jdo.option.RetainValues`

---

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

**Configuration API:** `kodo.conf.JDOConfiguration.getRetainValues`

**Resource adaptor config-property:** `RetainValues`

**Default:** `true`

**Description:** Whether persistent fields retain their values on transaction commit. See [Section 7.2, “PersistenceManagerFactory Properties” \[37\]](#) in the JDO Overview for details.

---

## 2.6. 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 alphabetical listing of Kodo JDO-specific properties is given below.

### 2.6.1. `kodo.AggregateListeners`

---

**Property name:** `kodo.AggregateListeners`

**Configuration API:** `kodo.conf.JDOConfiguration.getAggregateListeners`

**Resource adaptor config-property:** `AggregateListeners`

**Default:-**

**Description:** A comma-separated list of full plugin strings (see [Section 2.4, “Plugin Configuration” \[140\]](#)) for custom `kodo.query.AggregateListeners` to make available to all queries, in addition to the standard set of listeners. See [Section 10.7, “Query Extensions” \[290\]](#) for details on aggregates.

### 2.6.2. `kodo.ClassResolver`

---

**Property name:** `kodo.ClassResolver`

**Configuration API:** `kodo.conf.JDOConfiguration.getClassResolver`

**Resource adaptor config-property:** `ClassResolver`

**Default:** `spec`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.util.ClassResolver` implementation to use for class name resolution. The default value `spec` is an alias for the `kodo.util.SpecClassResolver`. This resolver is compliant with the candidate release of the JDO 1.0.2 specification, but you may wish to plug in your own implementations if you have special classloading needs.

### 2.6.3. `kodo.ConnectionProperties`

---

**Property name:** `kodo.ConnectionProperties`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionProperties`

**Resource adaptor config-property:** `ConnectionProperties`

**Default:** -

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) listing properties to configure the driver listed in the `ConnectionDriverName` property described in [Section 2.5.2, “`javax.jdo.option.ConnectionDriverName`” \[142\]](#). See [Chapter 4, \*JDBC\* \[166\]](#) for details.

### 2.6.4. `kodo.ConnectionFactoryProperties`

---

**Property name:** `kodo.ConnectionFactoryProperties`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionFactoryProperties`

**Resource adaptor config-property:** `ConnectionFactoryProperties`

**Default:** -

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) listing properties for configuration of the `javax.sql.DataSource` in use. See the [Chapter 4, JDBC \[166\]](#) for details.

---

## 2.6.5. kodo.Connection2DriverName

---

**Property name:** `kodo.Connection2DriverName`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnection2DriverName`

**Resource adaptor config-property:** `Connection2DriverName`

**Default:** -

**Description:** This property is equivalent to the `javax.jdo.option.ConnectionDriverName` property described in [Section 2.5.2, “javax.jdo.option.ConnectionDriverName” \[142\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[321\]](#) for details.

---

## 2.6.6. kodo.Connection2Password

---

**Property name:** `kodo.Connection2Password`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnection2Password`

**Resource adaptor config-property:** `Connection2Password`

**Default:** -

**Description:** This property is equivalent to the `javax.jdo.option.ConnectionPassword` property described in [Section 2.5.5, “javax.jdo.option.ConnectionPassword” \[142\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[321\]](#) for details.

---

## 2.6.7. kodo.Connection2URL

---

**Property name:** `kodo.Connection2URL`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnection2URL`

**Resource adaptor config-property:** `Connection2URL`

**Default:** -

**Description:** This property is equivalent to the `javax.jdo.option.ConnectionURL` property described in [Section 2.5.6, “javax.jdo.option.ConnectionURL” \[143\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[321\]](#) for details.

---

## 2.6.8. kodo.Connection2UserName

---

**Property name:** `kodo.Connection2UserName`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnection2UserName`

**Resource adaptor config-property:** `Connection2UserName`

**Default:** -

**Description:** This property is equivalent to the `javax.jdo.option.ConnectionUserName` property described in [Section 2.5.7, “`javax.jdo.option.ConnectionUserName`” \[143\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[32\]](#) for details.

## 2.6.9. `kodo.Connection2Properties`

---

**Property name:** `kodo.Connection2Properties`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnection2Properties`

**Resource adaptor config-property:** `Connection2Properties`

**Default:** -

**Description:** This property is equivalent to the `kodo.ConnectionProperties` property described in [Section 2.6.3, “`kodo.ConnectionProperties`” \[145\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[32\]](#) for details.

## 2.6.10. `kodo.ConnectionFactory2Properties`

---

**Property name:** `kodo.ConnectionFactory2Properties`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionFactory2Properties`

**Resource adaptor config-property:** `ConnectionFactory2Properties`

**Default:** -

**Description:** This property is equivalent to the `kodo.ConnectionFactoryProperties` property described in [Section 2.6.4, “`kodo.ConnectionFactoryProperties`” \[145\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[32\]](#) for details.

## 2.6.11. `kodo.ConnectionRetainMode`

---

**Property name:** `kodo.ConnectionRetainMode`

**Configuration API:** `kodo.conf.JDOConfiguration.getConnectionRetainMode`

**Resource adaptor config-property:** `ConnectionRetainMode`

**Default:** on-demand

**Description:** Controls how Kodo uses data store connections. This property can also be specified for individual persistence managers. See [Section 4.8, “Configuring the Use of JDBC Connections” \[177\]](#) for details.

## 2.6.12. `kodo.CopyObjectIds`

---

**Property name:** `kodo.CopyObjectIds`

**Configuration API:** `kodo.conf.JDOConfiguration.getCopyObjectIds`

**Resource adaptor config-property:** `CopyObjectIds`

**Default:** `false`

**Description:** Whether to copy object id values before returning them to your code from the `JDOHelper` or `PersistenceManager`. If you ever change object id values you obtain from Kodo, you should set this property to `true`.

## 2.6.13. kodo.DataCache

---

**Property name:** `kodo.DataCache`

**Configuration API:** `kodo.conf.JDOConfiguration.getDataCache`

**Resource adaptor config-property:** `DataCache`

**Default:** `false`

**Description:** A plugin list string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.datacache.DataCache` to use for data caching. See [Section 14.3.2, “Kodo JDO Cache Usage” \[329\]](#) for details.

## 2.6.14. kodo.DataCacheTimeout

---

**Property name:** `kodo.DataCacheTimeout`

**Configuration API:** `kodo.conf.JDOConfiguration.getDataCacheTimeout`

**Resource adaptor config-property:** `DataCacheTimeout`

**Default:** `-1`

**Description:** The number of milliseconds that data in the data cache is valid. Set this to -1 to indicate that data should not expire from the cache. This property can also be specified for individual classes. See [Section 14.3.2, “Kodo JDO Cache Usage” \[329\]](#) for details.

## 2.6.15. kodo.DynamicDataStructs

---

**Property name:** `kodo.DynamicDataStructs`

**Configuration API:** `kodo.conf.JDOConfiguration.getDynamicDataStructs`

**Resource adaptor config-property:** `DynamicDataStructs`

**Default:** `false`

**Description:** Whether to dynamically generate customized structs to hold persistent data. Both the Kodo data cache and the remote persistence manager framework rely on data structs to cache and transfer persistent state. With dynamic structs, Kodo can customize data storage for each class, eliminating the need to generate primitive wrapper objects. This saves memory and speeds up certain runtime operations. The price is a longer warm-up time for the application - generating and loading custom classes into the JVM takes time. Therefore, only set this property to `true` if you have a long-running application where the initial cost of class generation is offset by memory and speed optimization over time.

## 2.6.16. kodo.EagerFetchMode

---

**Property name:** `kodo.EagerFetchMode`

**Configuration API:** `kodo.conf.JDOConfiguration.getEagerFetchMode`

**Resource adaptor config-property:** `EagerFetchMode`

**Default:** `parallel`

**Description:** Optimizes how Kodo loads persistent relations. This property can also be specified on individual persistence manager, query, and extent instances. See [Section 14.2, “Eager Fetching” \[326\]](#) for details.

---

## 2.6.17. kodo.FetchBatchSize

---

**Property name:** `kodo.FetchBatchSize`

**Configuration API:** `kodo.conf.JDOConfiguration.getFetchBatchSize`

**Resource adaptor config-property:** `FetchBatchSize`

**Default:** `-1`

**Description:** The number of rows to fetch at once when scrolling through a result set. This property can also be specified for individual persistence manager, query, and extent instances. See [Section 4.10, “Large Result Sets” \[180\]](#) for details.

## 2.6.18. kodo.FetchGroups

---

**Property name:** `kodo.FetchGroups`

**Configuration API:** `kodo.conf.JDOConfiguration.getFetchGroups`

**Resource adaptor config-property:** `FetchGroups`

**Default:** `-`

**Description:** A comma-separated list of fetch group names that are to be loaded when loading objects from the data store. This property can also be set on individual persistence manager, query, and extent instances. See [Section 14.5, “Fetch Groups” \[337\]](#) for details.

## 2.6.19. kodo.FilterListeners

---

**Property name:** `kodo.FilterListeners`

**Configuration API:** `kodo.conf.JDOConfiguration.getFilterListeners`

**Resource adaptor config-property:** `FilterListeners`

**Default:** `-`

**Description:** A comma-separated list of full plugin strings (see [Section 2.4, “Plugin Configuration” \[140\]](#)) for custom `kodo.jdbc.query.JDBCFilterListeners` to make available to all queries, in addition to the standard set of listeners. You can also add filter listeners to individual Kodo JDO query instances. See [Section 10.7, “Query Extensions” \[290\]](#) for details.

## 2.6.20. kodo.FlushBeforeQueries

---

**Property name:** `kodo.FlushBeforeQueries`

**Configuration API:** `kodo.conf.JDOConfiguration.getFlushBeforeQueries`

**Resource adaptor config-property:** `FlushBeforeQueries`

**Default:** `with-connection`

**Description:** Whether or not to flush any changes made in the current transaction to the data store before executing a query. See [Section 4.8, “Configuring the Use of JDBC Connections” \[177\]](#) for details.

## 2.6.21. kodo.InverseManager

---

**Property name:** `kodo.InverseManager`

**Configuration API:** `kodo.conf.JDOConfiguration.getInverseManager`

**Resource adaptor config-property:** `InverseManager`

**Default:** `false`

**Possible values:** `false, true`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing a `kodo.runtime.InverseManager` to use for correcting inverse relations upon a flush. See [Section 5.3, “Managed Inverses” \[188\]](#) for usage documentation.

---

## 2.6.22. kodo.LicenseKey

**Property name:** `kodo.LicenseKey`

**Configuration API:** `kodo.conf.JDOConfiguration.getLicenseKey`

**Resource adaptor config-property:** `LicenseKey`

**Default:** `-`

**Description:** The license key provided to you by SolarMetric. Keys are available at [www.solarmetric.com](http://www.solarmetric.com).

---

## 2.6.23. kodo.LockManager

**Property name:** `kodo.LockManager`

**Configuration API:** `kodo.conf.JDOConfiguration.getLockManager`

**Resource adaptor config-property:** `LockManager`

**Default:** `pessimistic`

**Possible values:** `none, sjvm, pessimistic`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing a `kodo.runtime.LockManager` to use for acquiring locks on persistent instances during datastore transactions.

---

## 2.6.24. kodo.LockTimeout

**Property name:** `kodo.LockTimeout`

**Configuration API:** `kodo.conf.JDOConfiguration.getLockTimeout`

**Resource adaptor config-property:** `LockTimeout`

**Default:** `-1`

**Description:** The number of milliseconds to wait for an object lock before throwing an exception, or -1 for no limit. See [Section 10.8, “Object Locking” \[292\]](#) for details.

---

## 2.6.25. kodo.Log

**Property name:** `kodo.Log`

**Configuration API:** `kodo.conf.JDOConfiguration.getLog`

**Resource adaptor config-property:** Log

**Default:** true

**Possible values:** kodo, commons, log4j, none

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing a `com.solarmetric.log.LogFactory` to use for performing logging. For details on logging, see [Chapter 3, Logging \[161\]](#)

---

## 2.6.26. kodo.ManagedRuntime

**Property name:** `kodo.ManagedRuntime`

**Configuration API:** `kodo.conf.JDOConfiguration.getManagedRuntime`

**Resource adaptor config-property:** ManagedRuntime

**Default:** auto

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.ee.ManagedRuntime` implementation to use for obtaining a reference to the `TransactionManager` in an enterprise environment. See [Section 13.1, “Integrating with the Transaction Manager” \[321\]](#) for details.

---

## 2.6.27. kodo.ManagementConfiguration

**Property name:** `kodo.ManagementConfiguration`

**Configuration API:** `kodo.conf.JDOConfiguration.getManagementConfiguration`

**Resource adaptor config-property:** ManagementConfiguration

**Default:** none

**Description:** Allows for configuration of management and profiling capabilities. For more information, see [Chapter 12, Management and Monitoring \[309\]](#)

---

## 2.6.28. kodo.MetadataLoader

**Property name:** `kodo.MetadataLoader`

**Configuration API:** `kodo.conf.JDOConfiguration.getMetadataLoader`

**Resource adaptor config-property:** MetadataLoader

**Default:** jdo

**Description:** Allows for customization of how persistent object metadata is loaded.

---

## 2.6.29. kodo.ObjectLookupMode

**Property name:** `kodo.ObjectLookupMode`

**Configuration API:** `kodo.conf.JDOConfiguration.getObjectLookupMode`

**Resource adaptor config-property:** ObjectLookupMode

**Default:** check

**Description:** Determines Kodo's behavior on calls to `PersistenceManager.getObjectById` when the `validate` parameter is `false`. The default value, `check` checks that the object exists in the database and loads its fetch group fields. Set to `hollow` to return a hollow object without checking the database instead.

## 2.6.30. kodo.OrphanedKeyAction

---

**Property name:** `kodo.OrphanedKeyAction`

**Configuration API:** `kodo.conf.JDOConfiguration.getOrphanedKeyAction`

**Resource adaptor config-property:** `OrphanedKeyAction`

**Default:** `log`

**Possible values:** `log`, `exception`, `none`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing a `kodo.event.OrphanedKeyAction` to invoke when Kodo discovers an orphaned datastore key. See [Section 10.9, “Orphaned Keys” \[296\]](#) for details.

## 2.6.31. kodo.PersistenceManagerImpl

---

**Property name:** `kodo.PersistenceManagerImpl`

**Configuration API:** `kodo.conf.JDOConfiguration.getPersistenceManagerImpl`

**Resource adaptor config-property:** `PersistenceManagerImpl`

**Default:** `default`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `javax.jdo.PersistenceManager` type to use at runtime. See [Section 10.2.4, “PersistenceManager Extension” \[289\]](#) for details.

## 2.6.32. kodo.PersistenceManagerServer

---

**Property name:** `kodo.PersistenceManagerServer`

**Configuration API:** `kodo.conf.JDOConfiguration.getPersistenceManagerServer`

**Resource adaptor config-property:** `PersistenceManagerServer`

**Default:** `false`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing whether and how to service remote persistence managers with this persistence manager factory. See [Section 11.2, “Remote Persistence Managers” \[303\]](#) for details.

## 2.6.33. kodo.PersistentClasses

---

**Property name:** `kodo.PersistentClasses`

**Configuration API:** `kodo.conf.JDOConfiguration.getPersistentClasses`

**Resource adaptor config-property:** `PersistentClasses`

**Default:** -

**Description:** A comma-separated list of classes that are to be instantiated whenever a new `PersistenceManager` is created. See [Section 5.7, “Persistent Class List” \[196\]](#) for details.

## 2.6.34. kodo.ProxyManager

---

**Property name:** `kodo.ProxyManager`

**Configuration API:** `kodo.conf.JDOConfiguration.getProxyManager`

**Resource adaptor config-property:** `ProxyManager`

**Default:** `default`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing a `kodo.util.ProxyManager` to use for proxying mutable second class objects. See [Section 5.4.3.3, “Custom Proxies” \[191\]](#) for details.

## 2.6.35. kodo.QueryCache

---

**Property name:** `kodo.QueryCache`

**Configuration API:** `kodo.conf.JDOConfiguration.getQueryCache`

**Resource adaptor config-property:** `QueryCache`

**Default:** `true`, when the data cache (see [Section 2.6.13, “kodo.DataCache” \[148\]](#)) is also enabled, `false` otherwise.

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.datacache.QueryCache` implementation to use for caching of queries loaded from the data store. See [Section 14.3.3, “Query Caching” \[332\]](#) for details.

## 2.6.36. kodo.QueryCompilationCache

---

**Property name:** `kodo.QueryCompilationCache`

**Configuration API:** `kodo.conf.JDOConfiguration.getQueryCompilationCache`

**Resource adaptor config-property:** `QueryCompilationCache`

**Default:** `true`.

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `java.util.Map` to use for caching of data used during query compilation.

*Table 2.1. Pre-defined aliases*

Alias	Value
<code>true</code>	<code>kodo.util.CacheMap</code>
<code>all</code>	<code>java.util.HashMap</code>
<code>false</code>	<code>none</code>

## 2.6.37. kodo.ReadLockLevel

---

**Property name:** `kodo.ReadLockLevel`

**Configuration API:** `kodo.conf.JDOConfiguration.getReadLockLevel`

**Resource adaptor config-property:** `ReadLockLevel`

**Default:** `read`

**Possible values:** `none`, `read`, `write`, numeric values for lock-manager specific lock levels

**Description:** The default level at which to lock objects retrieved during a non-optimistic transaction. Note that for the default JD-BC lock manager, `read` and `write` lock levels are equivalent.

---

## 2.6.38. `kodo.RemoteCommitProvider`

**Property name:** `kodo.RemoteCommitProvider`

**Configuration API:** `kodo.conf.JDOConfiguration.getRemoteCommitProvider`

**Resource adaptor config-property:** `RemoteCommitProvider`

**Default:** `-`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.event.RemoteCommitProvider` implementation to use for distributed event notification. See [Section 14.4.1, “Remote Commit Provider Configuration” \[336\]](#) Remote Commit Provider Configuration for details.

---

## 2.6.39. `kodo.RestoreMutableValues`

**Property name:** `kodo.RestoreMutableValues`

**Configuration API:** `kodo.conf.JDOConfiguration.getRestoreMutableValues`

**Resource adaptor config-property:** `RestoreMutableValues`

**Default:** `false`

**Description:** Whether to restore mutable second class object fields such as collections and maps to their pre-transaction values when a rollback occurs. When setting this property to `true` you must also make sure the `javax.jdo.option.RestoreValues` property described in [Section 2.5.13, “`javax.jdo.option.RestoreValues`” \[144\]](#) is not set to `false`.

---

## 2.6.40. `kodo.RetainValuesInOptimistic`

**Property name:** `kodo.RetainValuesInOptimistic`

**Configuration API:** `kodo.conf.JDOConfiguration.getRetainValuesInOptimistic`

**Resource adaptor config-property:** `RetainValuesInOptimistic`

**Default:** `true`

**Description:** If `false`, then nontransactional objects will be cleared when they are first modified in an optimistic transaction. This cuts down on the possible number of optimistic verification exceptions, but it is not standard JDO behavior and can also have negative performance consequences depending on usage patterns.

---

## 2.6.41. `kodo.RetryClassRegistration`

**Property name:** `kodo.RetryClassRegistration`

**Configuration API:** `kodo.conf.JDOConfiguration.getRetryClassRegistration`

**Resource adaptor config-property:** `RetryClassRegistration`

**Default:** `false`

**Description:** Controls whether to log a warning and defer registration instead of throwing an exception when a class registered with JDO cannot be fully processed. This property should *only* be used in complex classloader situations where security is preventing Kodo from reading registered classes. Setting this to true unnecessarily may obscure more serious problems.

---

## 2.6.42. `kodo.SubclassFetchMode`

**Property name:** `kodo.SubclassFetchMode`

**Configuration API:** `kodo.conf.JDOConfiguration.getSubclassFetchMode`

**Resource adaptor config-property:** `SubclassFetchMode`

**Default:** `parallel`

**Description:** How to select subclass data when it is in other tables. This property can also be set on individual persistence manager, query, and extent instances. See [Section 14.2, “Eager Fetching” \[326\]](#)

---

## 2.6.43. `kodo.TransactionMode`

**Property name:** `kodo.TransactionMode`

**Configuration API:** `kodo.conf.JDOConfiguration.getTransactionMode`

**Resource adaptor config-property:** `TransactionMode`

**Default:** `local`

**Description:** The default transaction mode to use when obtaining a persistence manager. This property can also be specified when obtaining an individual persistence manager. See [Section 13.1, “Integrating with the Transaction Manager” \[324\]](#) for details.

---

## 2.6.44. `kodo.WriteLockLevel`

**Property name:** `kodo.WriteLockLevel`

**Configuration API:** `kodo.conf.JDOConfiguration.getWriteLockLevel`

**Resource adaptor config-property:** `WriteLockLevel`

**Default:** `write`

**Possible values:** `none`, `read`, `write`, numeric values for lock-manager specific lock levels

**Description:** The default level at which to lock objects changed during a non-optimistic transaction. Note that for the default JDBC lock manager, `read` and `write` lock levels are equivalent.

---

## 2.6.45. `kodo.jdbc.ClassIndicator`

**Property name:** `kodo.jdbc.ClassIndicator`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getClassIndicator`

**Resource adaptor config-property:** `ClassIndicator`

**Default:** `in-class-name`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the default `kodo.jdbc.meta.ClassIndicator` to install on new mappings. See [Section 7.8, “Class Indicator” \[232\]](#) for details.

---

## 2.6.46. kodo.jdbc.ConnectionDecorators

**Property name:** `kodo.jdbc.ConnectionDecorators`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getConnectionDecorators`

**Resource adaptor config-property:** `ConnectionDecorators`

**Default:** -

**Description:** A comma-separated list of plugin strings (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing `com.solarmetric.jdbc.ConnectionDecorator` instances to install on the connection factory. These decorators can wrap connections passed from the underlying data source to add functionality. Kodo will pass all connections through the list of decorators before using them. Note that by default Kodo JDO employs all of the built-in decorators in the `com.solarmetric.jdbc` package already; you do not need to list them here.

---

## 2.6.47. kodo.jdbc.DataSourceMode

**Property name:** `kodo.jdbc.DataSourceMode`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getDataSourceMode`

**Resource adaptor config-property:** `DataSourceMode`

**Default:** `local`

**Description:** The data source mode to use when integrating with the application server's managed transactions. See [Section 4.2.1, “Enlisted Data Sources” \[169\]](#) for details.

---

## 2.6.48. kodo.jdbc.DBDictionary

**Property name:** `kodo.jdbc.DBDictionary`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getDBDictionary`

**Resource adaptor config-property:** `DBDictionary`

**Default:** Based on the `javax.jdo.option.ConnectionURL` (see [Section 2.5.6, “javax.jdo.option.ConnectionURL” \[143\]](#)) and `javax.jdo.option.ConnectionDriverName` (see [Section 2.5.2, “javax.jdo.option.ConnectionDriverName” \[142\]](#)).

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.jdbc.sql.DBDictionary` to use for database interaction. Kodo JDO typically auto-configures the dictionary based on the JDBC URL, but you may have to set this property explicitly if you are using an unrecognized driver, or to plug in your own dictionary for a database Kodo JDO does not support out-of-the-box. See [Section 4.3, “Database Support” \[169\]](#) for details.

---

## 2.6.49. kodo.jdbc.FetchDirection

**Property name:** `kodo.jdbc.FetchDirection`

**Configuration API:** `kodo.conf.JDBCCConfiguration.getFetchDirection`

**Resource adaptor config-property:** `FetchDirection`

**Default:** `forward`

**Description:** The expected order in which query result lists will be accessed. This property can also be specified for individual persistence manager, query, and extent instances. See [Section 4.10, “Large Result Sets” \[180\]](#) for details.

---

## 2.6.50. kodo.jdbc.ForeignKeyConstraints

---

**Property name:** `kodo.jdbc.ForeignKeyConstraints`

**Configuration API:** `kodo.conf.JDBCCConfiguration.getForeignKeyConstraints`

**Resource adaptor config-property:** `ForeignKeyConstraints`

**Default:** `false`

**Description:** Whether to evaluate database foreign key constraints and order all SQL operations so that referential integrity is not violated. See [Section 4.11, “SQL Statement Ordering & Foreign Keys” \[182\]](#) in SQL Statement Ordering for details.

---

## 2.6.51. kodo.jdbc.JDBCListeners

---

**Property name:** `kodo.jdbc.JDBCListeners`

**Configuration API:** `kodo.jdbc.conf.JDBCCConfiguration.getJDBCListeners`

**Resource adaptor config-property:** `JDBCListeners`

**Default:** `-`

**Description:** A comma-separated list of plugin strings (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing `com.solarmetric.jdbc.JDBCListener` event listeners to install. These listeners will be notified on various JDBC-related events. The `com.solarmetric.jdbc.PerformanceTracker` is one such listener that can be used to track JDBC performance.

---

## 2.6.52. kodo.jdbc.LRSSize

---

**Property name:** `kodo.jdbc.LRSSize`

**Configuration API:** `kodo.conf.JDBCCConfiguration.getLRSSize`

**Resource adaptor config-property:** `LRSSize`

**Default:** `query`

**Description:** The strategy to use to calculate the size of a result list. This property can also be specified for individual persistence manager, query, and extent instances. See [Section 4.10, “Large Result Sets” \[180\]](#) for details.

---

## 2.6.53. kodo.jdbc.MappingFactory

---

**Property name:** `kodo.jdbc.MappingFactory`

**Configuration API:** `kodo.jdbc.conf.JDBCCConfiguration.getMappingFactory`

**Resource adaptor config-property:** `MappingFactory`

**Default:** `file`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.jdbc.meta.MappingFactory` to use to store and retrieve object-relational mapping information for your persistent classes. See [Section 7.3, “Mapping Factory” \[214\]](#) for details.

## 2.6.54. kodo.jdbc.ResultSetType

---

**Property name:** `kodo.jdbc.ResultSetType`

**Configuration API:** `kodo.conf.JDBCConfiguration.getResultSetType`

**Resource adaptor config-property:** `ResultSetType`

**Default:** `forward-only`

**Description:** The JDBC result set type to use when fetching result lists. This property can also be specified for individual persistence manager, query, and extent instances. See [Section 4.10, “Large Result Sets” \[180\]](#) for details.

## 2.6.55. kodo.jdbc.SchemaFactory

---

**Property name:** `kodo.jdbc.SchemaFactory`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getSchemaFactory`

**Resource adaptor config-property:** `SchemaFactory`

**Default:** `native`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `kodo.jdbc.schema.SchemaFactory` to use to store and retrieve information about the database schema. See [Section 8.1.2, “Schema Factory” \[278\]](#) for details.

## 2.6.56. kodo.jdbc.Schemas

---

**Property name:** `kodo.jdbc.Schemas`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getSchemas`

**Resource adaptor config-property:** `Schemas`

**Default:** `-`

**Description:** A comma-separated list of the schemas and/or tables used for your persistent JDO-related data. See [Section 8.1.1, “Schemas List” \[278\]](#) for details.

## 2.6.57. kodo.jdbc.SequenceFactory

---

**Property name:** `kodo.jdbc.SequenceFactory`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getSequenceFactory`

**Resource adaptor config-property:** `SequenceFactory`

**Default:** `db`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the

`kodo.jdbc.schema.SequenceFactory` to use to generate datastore identity values. See [Section 5.2.4.1, “Sequence Factory”](#) [185] for details.

## 2.6.58. kodo.jdbc.SubclassMapping

---

**Property name:** `kodo.jdbc.SubclassMapping`

**Configuration API:** `kodo.jdbc.conf.JDBCCConfiguration.getSubclassMapping`

**Resource adaptor config-property:** `SubclassMapping`

**Default:** `flat`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration”](#) [140]) describing the default `kodo.jdbc.meta.ClassMapping` to install on new subclass mappings. See [Section 7.6, “Class Mapping”](#) [219] for details.

## 2.6.59. kodo.jdbc.SynchronizeMappings

---

**Property name:** `kodo.jdbc.SynchronizeMappings`

**Configuration API:** `kodo.conf.JDBCCConfiguration.getSynchronizeMappings`

**Resource adaptor config-property:** `SynchronizeMappings`

**Default:** `false`

**Description:** Controls whether Kodo will attempt to run the mapping tool on all persistent classes to synchronize their mappings and schema at runtime. Useful for rapid test/debug cycles. See [Section 7.2, “Automatic Runtime Mapping”](#) [213] for more information.

## 2.6.60. kodo.jdbc.TransactionIsolation

---

**Property name:** `kodo.jdbc.TransactionIsolation`

**Configuration API:** `kodo.conf.JDBCCConfiguration.getTransactionIsolation`

**Resource adaptor config-property:** `TransactionIsolation`

**Default:** `default`

**Description:** The JDBC transaction isolation level to use. See [Section 4.6, “Setting the Transaction Isolation”](#) [176] for details.

## 2.6.61. kodo.jdbc.UpdateManager

---

**Property name:** `kodo.jdbc.UpdateManager`

**Configuration API:** `kodo.jdbc.conf.JDBCCConfiguration.getUpdateManager`

**Resource adaptor config-property:** `UpdateManager`

**Default:** `default`

**Description:** The full class name of the `kodo.jdbc.runtime.UpdateManager` to use to flush persistent object changes to the data store. The provided default implementation ( `kodo.jdbc.runtime.UpdateManagerImpl` , with a configuration alias of `default` ) will suit all but the most advanced users.

## 2.6.62. kodo.jdbc.VersionIndicator

---

**Property name:** `kodo.jdbc.VersionIndicator`

**Configuration API:** `kodo.jdbc.conf.JDBCConfiguration.getVersionIndicator`

**Resource adaptor config-property:** `VersionIndicator`

**Default:** `version-number`

**Description:** A plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the default `kodo.jdbc.meta.VersionIndicator` to install on new mappings. See [Section 7.7, “Version Indicator” \[229\]](#) for details.

---

# Chapter 3. Logging

Logging is 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 can assist in quickly identifying any misconfigurations in the mapping information. Kodo provides a flexible logging system that integrates with many existing runtime systems, such as application servers and servlet runners.

There are four built-in log plugins: a **simple logging framework** that covers basic needs, a **Log4J** delegate, an **Apache Commons Logging** delegate, and a **no-op** implementation for disabling logging.

## Warning

Logging can have a negative impact on performance. 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. You can disable logging altogether by setting the `kodo.Log` property to `none`.

## 3.1. Logging Channels

---

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. Kodo uses the following logging channels:

- `kodo.Tool`: Messages issued by the Kodo tools when run on the command line or via Ant. Most messages are basic statements detailing which classes or files the tools are running on. Detailed output is only available via the logging category the tool belongs to, such as `kodo.Enhance` for the enhancer (see [Section 5.5, “Enhancement” \[19\]](#)) or `kodo.Metadata` for the mapping tool (see [Section 7.1, “Mapping Tool” \[20\]](#)). This logging category is provided so that you can get a general idea of what a tool is doing without having to manipulate logging settings that might also affect runtime behavior.
- `kodo.Configuration`: Messages issued by the configuration framework.
- `kodo.Enhance`: Messages issued by the JDO enhancement process.
- `kodo.Metadata`: Details about the parsing of JDO metadata and object-relational data.
- `kodo.Runtime`: General Kodo runtime messages.
- `kodo.Query`: Messages about queries issued via the JDO query facilities. Queries and any parameter values, if applicable, will be logged to the `TRACE` level at execution time. Information about possible performance concerns will be logged to the `INFO` level.
- `kodo.Remote`: Remote connection and execution messages.
- `kodo.DataCache`: Messages from the L2 data cache plugins.
- `kodo.jdbc.JDBC`: JDBC connection information. General JDBC information will be logged to the `TRACE` level. Information about possible performance concerns will be logged to the `INFO` level.
- `kodo.jdbc.SQL`: This is the most common logging channel to use. Detailed information about the execution of SQL statements will be sent to the `TRACE` level. It is useful to enable this channel if you are curious about the exact SQL that Kodo issues to the data store.

When using the built-in Kodo logging facilities, you can enable this by adding `SQL=TRACE` to your `kodo.Log` property:

```
kodo.Log: SQL=TRACE
```

Kodo can optionally reformat the logged SQL to make it easier to read. To enable pretty-printing, add `PrettyPrint=true` to the `kodo.ConnectionFactoryProperties` property. You can control how many columns wide the pretty-printed SQL will be with the `PrettyPrintLineLength` property. The default line length is 60 columns.

While pretty printing makes things easier to read, it can make it harder to process with tools like `grep`.

Pretty-printing properties configuration might look like so:

```
kodo.ConnectionFactoryProperties: MaxActive=100, PrettyPrint=true, PrettyPrintLineLength=72
```

- `kodo.jdbc.Schema`: Details about operations on the database schema.
- `com.solarmetric.Manage`: JMX and management-related logging channel.
- `com.solarmetric.Profile`: Information related to Kodo's profiling framework.

## 3.2. Kodo Logging

By default, Kodo uses a basic logging framework with the following output format:

*millis level [thread name] channel - message*

For example, when loading an application that uses Kodo, a message like the following will be sent to the `kodo.Metadata` log channel when Kodo loads JDO metadata about your project:

```
2107 INFO [main] kodo.Metadata - Parsing metadata resource "file:/projects/JDO/test/my/company/package.jdo".
```

The default logging system accepts the following parameters:

- `File`: The name of the file to log to, or `stdout` or `stderr` to send messages to standard out and standard error, respectively. By default, Kodo sends log messages to standard error.
- `DefaultLevel`: The default logging level of unconfigured channels. Recognized values are `TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR`. Defaults to `INFO`.
- `DiagnosticContext`: A string that should be put in the beginning of all log messages issued by this `PersistenceManagerFactory`.
- `<channel>`: Using the last token of the **logging channel** name, you can configure the log level to use for that channel. See the examples below.

### Example 3.1. Standard Kodo Log Configuration

```
kodo.Log: DefaultLevel=WARN, Runtime=INFO, Tool=INFO
```

***Example 3.2. Standard Kodo Log Configuration + All SQL Statements***

```
kodo.Log: DefaultLevel=WARN, Runtime=INFO, Tool=INFO, SQL=TRACE
```

***Example 3.3. Logging to a File***

```
kodo.Log: File=/tmp/kodo.log, DefaultLevel=WARN, Runtime=INFO, Tool=INFO
```

## 3.3. 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 `kodo.Log` property to `none`:

```
kodo.Log: none
```

However, disabling logging permanently will cause all warnings to be consumed. So, we recommend using one of the more sophisticated mechanisms described in this chapter.

## 3.4. Log4J

---

When `kodo.Log` is set to `log4j`, Kodo will use Log4J for logging needs:

```
kodo.Log: log4j
```

In a standalone application, Log4J 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` directories). When deploying to a web or EJB application server, Log4J configuration is often performed in a `log4j.xml` file instead of a properties file. For further details on configuring Log4J, please see the [Log4J Manual](#). We present several example `log4j.properties` files below.

***Example 3.4. Standard Log4J Logging***

```
log4j.rootCategory=WARN, console
log4j.category.kodo.Tool=INFO
log4j.category.kodo.Runtime=INFO
log4j.category.kodo.Remote=WARN
log4j.category.kodo.DataCache=WARN
log4j.category.kodo.Metadata=WARN
```

```
log4j.category.kodo.Enhance=WARN
log4j.category.kodo.Query=WARN
log4j.category.kodo.jdbc.SQL=WARN
log4j.category.kodo.jdbc.JDBC=WARN
log4j.category.kodo.jdbc.Schema=WARN

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

## 3.5. Apache Commons Logging

Kodo JDO can use 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 simple console logging.

```
kodo.Log: commons
```

When using the Commons Logging framework in conjunction with Log4J, configuration will be the same as was discussed in the Log4J section above.

### 3.5.1. JDK 1.4 java.util.logging

When using JDK 1.4 or higher in conjunction with Kodo's Commons Logging support, logging will proceed through the built-in logging package provided by the **java.util.logging** package. 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. For example:

```
java -Djava.util.logging.config.file=mylogging.properties com.company.MyClass
```

#### *Example 3.5. JDK 1.4 Log Properties*

```
# 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
kodo.Tool.level=INFO
kodo.Runtime.level=INFO
kodo.Remote.level=INFO
kodo.DataCache.level=INFO
```

```
kodo.Metadata.level=INFO
kodo.Enhance.level=INFO
kodo.Query.level=INFO
kodo.jdbc.SQL.level=INFO
kodo.jdbc.JDBC.level=INFO
kodo.jdbc.Schema.level=INFO
```

## 3.6. Custom Log

If none of available logging systems meet your needs, the logging system can be configured to use a custom logging class. You might use custom logging to integrate with a proprietary logging framework used by some applications servers, or for logging to a graphical component for GUI applications.

A custom logging framework must include an implementation of the **com.solarmetric.log.LogFactory** interface. We present a custom log class example below.

### *Example 3.6. Custom Logging Class*

```
package com.xyz;

import com.solarmetric.log.*;

public class CustomLogFactory
    implements LogFactory
{
    private String _prefix = "CUSTOM LOG";

    public void setPrefix (String prefix)
    {
        _prefix = prefix;
    }

    public Log getLog (String channel)
    {
        // return a simple extension of AbstractLog that will log
        // everything to the System.err stream. Note that this is
        // roughly equivalent to Kodo's default logging behavior.
        return new AbstractLog ()
        {
            protected boolean isEnabled (short logLevel)
            {
                // log all levels
                return true;
            }

            protected void log (short type, String message, Throwable t)
            {
                // just send everything to System.err
                System.err.println (_prefix + ": " + type + ": "
                    + message + ": " + t);
            }
        };
    }
}
```

To make Kodo use your custom log factory, set the **kodo.Log** configuration property to your factory's full class name. Because this property is a plugin property (see [Section 2.4, “Plugin Configuration” \[140\]](#)), you can also pass parameters to your factory. For example, to use the example factory above and set its prefix to "LOG MSG", you could use the following setting:

```
kodo.Log: com.xyz.CustomLogFactory(Prefix="LOG MSG")
```

---

# Chapter 4. JDBC

Kodo JDO uses a relational database for object persistence. It communicates with the database using the Java DataBase Connectivity (JDBC) APIs. This chapter describes how to configure Kodo to work with the JDBC driver for your database, and how to access JDBC functionality at runtime.

## 4.1. Using the Kodo JDO DataSource

---

Kodo JDO includes its own `javax.sql.DataSource` implementation, complete with configurable connection pooling, SQL logging, and prepared statement caching. If you choose to use Kodo JDO's `DataSource`, then you must specify the properties below.

Required properties:

- `javax.jdo.option.ConnectionUserName`: The JDBC user name for connecting to the database.
- `javax.jdo.option.ConnectionPassword`: The JDBC password for the above user.
- `javax.jdo.option.ConnectionURL`: The JDBC URL for the database.
- `javax.jdo.option.ConnectionDriverName`: The JDBC driver.

To configure advanced features such as connection pooling and prepared statement caching, or to configure the underlying JDBC driver, use the following optional properties. The syntax of these property strings follows the syntax of Kodo plugin parameters described in [Section 2.4, “Plugin Configuration” \[140\]](#).

- **kodo.ConnectionProperties**: If the listed driver is an instance of `java.sql.Driver`, this string will be parsed into a `Properties` instance, which will then be used to obtain database connections through the `java.sql.Driver.connect(String url, Properties props)` method. If, on the other hand, the driver is a `javax.sql.DataSource`, the string will be treated as a plugin properties string, and matched to the bean setter methods of the `DataSource` instance.
- **kodo.ConnectionFactoryProperties**: Kodo JDO's built-in `DataSource` has the following configuration options you can set via this plugin string:
  - **ExceptionAction**: The action to take when when a connection that has thrown an exception is returned to the pool. Set to `destroy` to destroy the connection. Set to `validate` to validate the connection (subject to the `TestOnReturn`, `TestOnBorrow`, and other test settings). Set to `none` to ignore the fact that the connection has thrown an exception, and assume it is still usable. Defaults to `destroy`.
  - **MaxActive**: The maximum number of database connections in use at one time. A value of 0 disables connection pooling. Defaults to 8. This is the maximum number of connections that Kodo will give out to your application. If a connection is requested while `MaxActive` other connections are in use, Kodo will wait for `MaxWait` milliseconds for a connection to be returned, and will then throw an exception if no connection was made available.
  - **MaxIdle**: The maximum number of idle database connections to keep in the pool. Defaults to 8. If this number is less than `MaxActive`, then Kodo will close extra connections that are returned to the pool if there are already `MaxIdle` available connections in the pool. This allows for unexpected or atypical load while still maintaining a relatively small pool when there is less load on the system.
  - **MaxTotal**: The maximum number of database connections in the pool, whether active or idle. Defaults to -1, meaning no limit (limit will be controlled by `MaxActive` and `MaxIdle` for each login criteria).

- **MaxWait:** The maximum number of milliseconds to wait for a free database connection to become available before giving up. Defaults to 3000.
- **MinEvictableIdleTimeMillis:** The minimum number of milliseconds that a database connection can sit idle before it becomes a candidate for eviction from the pool. Defaults to 30 minutes. Set to 0 to never evict a connection based on idle time alone.
- **RollbackOnReturn:** Force all connections to be rolled back when they are returned to the pool. If false, the datasource will only roll back connections when it detects that there has been any transactional activity on the connection.
- **TestOnBorrow:** Whether to validate database connections before obtaining them from the pool. Note that validation only consists of a call to the connection's `isClosed` method unless you specify a `ValidationSQL` string to use to send a quick query. Defaults to `true`.
- **TestOnReturn:** Set to `true` to validate database connections when they are returned to the pool. Note that validation only consists of a call to the connection's `isClosed` method unless you specify a `ValidationSQL` string to use to send a quick query.
- **TestWhileIdle:** Set to `true` to periodically validate idle database connections.
- **TimeBetweenEvictionRunsMillis:** The number of milliseconds between runs of the eviction thread. Defaults to -1, meaning the eviction thread will never run.
- **ValidationSQL:** A simple SQL query to issue to validate a database connection. If this property is not set, then the only validation performed is to use the `Connection.isClosed` method. The following table shows the default settings for different databases. If a database is not shown, this property defaults to null.

**Table 4.1. Validation SQL Defaults**

Database	SQL
DB2	SELECT DISTINCT(CURRENT TIMESTAMP) FROM SYSIBM.SYSTABLES
Empress	SELECT DISTINCT(TODAY) FROM SYS_TABLES
Informix	SELECT DISTINCT CURRENT TIMESTAMP FROM INFORMIX.SYSTABLES
MySQL	SELECT NOW()
Oracle	SELECT SYSDATE FROM DUAL
Postgres	SELECT NOW()
SQLServer	SELECT GETDATE()
Sybase	SELECT GETDATE()

To disable validation SQL, set this property to an empty string, as in **Example 4.1, “Properties File for the Kodo JDO DataSource” [168]**

- **ValidationTimeout:** The minimum number of milliseconds that must elapse before a connection will ever be re-validated. This property is usually used with `TestOnBorrow` or `TestOnReturn` to reduce the number of validations performed, because the same connection is often borrowed and returned many times in short periods of time. Defaults to 300000 (5 minutes).
- **WhenExhaustedAction:** The action to take when there are no available database connections in the pool. Set to `exception` to immediately throw an exception. Set to `block` to block until a connection is available or the maximum wait time is exceeded. Set to `grow` to automatically grow the pool. Defaults to `block`.

Additionally, the following properties are available whether you use Kodo JDO's built-in `DataSource` or a third-party's:

- **MaxCachedStatements**: The maximum number of `java.sql.PreparedStatements` to cache. Statement caching can dramatically speed up some databases. Defaults to 50 for Kodo's data source, and 0 for third-party data sources. Most third-party data sources do not benefit from Kodo's prepared statement cache, because each returned connection has a unique hash code, making it impossible for Kodo to match connections to their cached statements.
- **QueryTimeout**: The maximum number of seconds the JDBC driver will wait for a statement to execute.

### *Example 4.1. Properties File for the Kodo JDO DataSource*

```
javax.jdo.option.ConnectionUserName: user
javax.jdo.option.ConnectionPassword: pass
javax.jdo.option.ConnectionURL: jdbc:hsqldb:db-hypersonic
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
kodo.ConnectionFactoryProperties: MaxActive=50, MaxIdle=10, \
    ValidationTimeout=50000, MaxCachedStatements=100, ValidationSQL=""
```

## 4.2. Using a Third-Party DataSource

---

Kodo JDO can be used with any third-party `javax.sql.DataSource`. There are 2 primary ways of telling Kodo JDO about a `DataSource`:

- Bind the `DataSource` into JNDI, and then specify its location in the **`javax.jdo.option.ConnectionFactoryName`** property.
- Specify the full class name of the `DataSource` implementation in the **`javax.jdo.option.ConnectionDriverName`** property in place of a JDBC driver. In this configuration Kodo JDO will instantiate an instance of the named class via reflection. It will then configure the `DataSource` with the properties in the **`kodo.ConnectionProperties`** setting.

Some advanced features of Kodo JDO's own `DataSource` can also be used with third-party implementations. Kodo JDO layers on top of the third-party `DataSource` to provide the extra functionality. To configure these advanced features, including prepared statement caching, use the **`kodo.ConnectionFactoryProperties`** property described in the previous section.

### *Example 4.2. Properties File for a Third-Party DataSource*

```
javax.jdo.option.ConnectionDriverName: oracle.jdbc.pool.OracleDataSource
kodo.ConnectionProperties: PortNumber=1521, ServerName=saturn, \
    DatabaseName=solarsid, DriverType=thin
kodo.ConnectionFactoryProperties: QueryTimeout=5000
```

## 4.2.1. Enlisted Data Sources

Certain application servers automatically enlist their data sources in global transactions. When this is the case, Kodo should not attempt to commit the underlying connection, leaving JDBC transaction completion to the application server. To notify Kodo that your third-party data source will automatically be enlisted in transactions, set the `kodo.jdbc.DataSourceMode` property to enlisted:

```
kodo.jdbc.DataSourceMode: enlisted
```

Note that Kodo can only use enlisted data sources when it is also integrating with the application server's managed transactions, as discussed in [Section 13.1, “Integrating with the Transaction Manager” \[321\]](#). Also note that all XA data sources are enlisted data sources, and you must set this property when using an XA data source. XA transactions are detailed in [Section 13.2, “XA Transactions” \[321\]](#)

When using an enlisted datasource, you should also configure a second non-enlisted datasource that Kodo can use to perform tasks that are independent of the global transaction in progress. The most common of these tasks is updating the sequence table Kodo uses to generate unique identity values for your datastore identity objects. Configure the second data source just as the first, but use the various "2" connection properties, such as `javax.jdo.option.ConnectionFactory2Name` or `javax.jdo.option.Connection2DriverName`. These properties are outlined in [Chapter 2, Configuration \[139\]](#). Typically, you will use Kodo's built-in datasource for the second data source (see [Section 4.1, “Using the Kodo JDO DataSource” \[166\]](#)).

## 4.3. Database Support

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. Typically, Kodo JDO auto-configures its JDBC behavior and SQL for your database, based on the values of your connection-related JDO configuration properties.

If Kodo JDO cannot detect what type of database you are using, or if you are using an unsupported database, you will have to tell Kodo JDO what `kodo.jdbc.sql.DBDictionary` to use. The `DBDictionary` abstracts away the differences between databases. You can plug a dictionary into Kodo JDO using the `kodo.jdbc.DBDictionary` configuration property. The built-in dictionaries are listed below. If you are using an unsupported database, you may have to write your own `DBDictionary` subclass, a simple process.

- `access`: Dictionary for Microsoft Access. This is an alias for the `kodo.jdbc.sql.AccessDictionary` class.
- `db2`: Dictionary for IBM's DB2 database. This is an alias for the `kodo.jdbc.sql.DB2Dictionary` class.
- `derby`: Dictionary for the Apache Derby database. This is an alias for the `kodo.jdbc.sql.DerbyDictionary` class.
- `empress`: Dictionary for Empress database. This is an alias for the `kodo.jdbc.sql.EmpressDictionary` class.
- `foxpro`: Dictionary for Microsoft Visual FoxPro. This is an alias for the `kodo.jdbc.sql.FoxProDictionary` class.
- `hsqldb`: Dictionary for the Hypersonic SQL database. This is an alias for the `kodo.jdbc.sql.HSQLDictionary` class.
- `informix`: Dictionary for the Informix database. This is an alias for the `kodo.jdbc.sql.InformixDictionary` class.
- `jdatastore`: Dictionary for Borland JDataStore. This is an alias for the `kodo.jdbc.sql.JDataStoreDictionary` class.
- `mysql`: Dictionary for the MySQL database. This is an alias for the `kodo.jdbc.sql.MySQLDictionary` class.

- `oracle`: Dictionary for Oracle. This is an alias for the `kodo.jdbc.sql.OracleDictionary` class.
- `pointbase`: Dictionary for Pointbase Embedded database. This is an alias for the `kodo.jdbc.sql.PointbaseDictionary` class.
- `postgres`: Dictionary for PostgreSQL. This is an alias for the `kodo.jdbc.sql.PostgresDictionary` class.
- `sqlserver`: Dictionary for Microsoft's SQLServer database. This is an alias for the `kodo.jdbc.sql.SQLServerDictionary` class.
- `sybase`: Dictionary for Sybase. This is an alias for the `kodo.jdbc.sql.SybaseDictionary` class.

The standard dictionaries all recognize the following properties. These properties will usually not need to be overridden, since the dictionary implementation should use the appropriate default values for your database. Typically these properties will only be changed when designing your own `DBDictionary` for an unsupported database.

- `DriverVendor`: The vendor of the particular JDBC driver you are using. Some dictionaries must alter their behavior depending on the driver vendor. See the `VENDOR_XXX` constants defined in your dictionary's Javadoc for available options.
- `CatalogSeparator`: The string the database uses to delimit between the schema name and the table name. This is typically `" . "`, which is the default.
- `CreatePrimaryKeys`: If `false`, then do not create database primary keys for identifiers. Defaults to `true`.
- `ConstraintNameMode`: When creating constraints, whether to put the constraint name before the definition (`before`), just after the constraint type name (`mid`), or after the constraint definition (`after`). Defaults to `before`.
- `MaxTableNameLength`: The maximum number of characters in a table name. Defaults to 128.
- `MaxColumnNameLength`: The maximum number of characters in a column name. Defaults to 128.
- `MaxPrimaryKeyNameLength`: The maximum number of characters in a primary key name. Defaults to 128.
- `MaxForeignKeyNameLength`: The maximum number of characters in a foreign key name. Defaults to 128.
- `MaxIndexNameLength`: The maximum number of characters in an index name. Defaults to 128.
- `MaxAutoIncrementNameLength`: Set this property to the maximum length of name for sequences used for auto-increment columns. Names longer than this value are truncated. Defaults to 31.
- `MaxIndexesPerTable`: The maximum number of indexes that can be placed on a single table. Defaults to no limit.
- `SupportsForeignKeys`: Whether the database supports foreign keys. Defaults to `true`.
- `SupportsDeferredConstraints`: Whether the database supports deferred constraints. Defaults to `true`.
- `SupportsRestrictDeleteAction`: Whether the database supports the `RESTRICT` foreign key delete action. Defaults to `true`.
- `SupportsCascadeDeleteAction`: Whether the database supports the `CASCADE` foreign key delete action. Defaults to `true`.
- `SupportsNullDeleteAction`: Whether the database supports the `SET NULL` foreign key delete action. Defaults to `true`.
- `SupportsDefaultDeleteAction`: Whether the database supports the `SET DEFAULT` foreign key delete action. Defaults to `true`.

- `SupportsAlterTableWithAddColumn`: Whether the database supports adding a new column in an ALTER TABLE statement. Defaults to `true`.
- `SupportsAlterTableWithDropColumn`: Whether the database supports dropping a column in an ALTER TABLE statement. Defaults to `true`.
- `ReservedWords`: A comma-separated list of reserved words for this database, beyond the standard SQL92 keywords.
- `SystemTables`: A comma-separated list of table names that should be ignored.
- `SystemSchemas`: A comma-separated list of schema names that should be ignored.
- `SchemaCase`: The case to use when querying the database metadata about schema components. Defaults to making all names upper case. Available values are: `upper`, `lower`, `preserve`.
- `ValidationSQL`: The SQL used to validate that a connection is still in a valid state. For example, "SELECT SYSDATE FROM DUAL" for Oracle.
- `InitializationSQL`: A piece of SQL to issue against the database whenever a connection is retrieved from the data source.
- `JoinSyntax`: The SQL join syntax to use in select statements. The available settings are:
  - `traditional`: Traditional SQL join syntax; outer joins are not supported.
  - `database`: The database's native join syntax. Databases that do not have a native syntax will default to one of the other options.
  - `sql92`: ANSI SQL92 join syntax. Outer joins are supported. Not all databases support this syntax.
- `CrossJoinClause`: The clause to use for a cross join (cartesian product). Defaults to `CROSS JOIN`.
- `InnerJoinClause`: The clause to use for an inner join. Defaults to `INNER JOIN`.
- `OuterJoinClause`: The clause to use for an left outer join. Defaults to `LEFT OUTER JOIN`.
- `RequiresConditionForCrossJoin`: Some databases require that there always be a conditional statement for a cross join. If set, this parameter ensures that there will always be some condition to the join clause.
- `ForUpdateClause`: The clause to append to SELECT statements to issue queries that obtain pessimistic locks. Defaults to `FOR UPDATE`.
- `TableForUpdateClause`: The clause to append to the end of each table alias in queries that obtain pessimistic locks. Defaults to `null`.
- `ToUpperCaseFunction`: SQL function call for for converting a string to upper case. Use the token `{0}` to represent the argument.
- `ToLowerCaseFunction`: Name of the SQL function for converting a string to lower case. Use the token `{0}` to represent the argument.
- `StringLengthFunction`: Name of the SQL function for getting the length of a string. Use the token `{0}` to represent the argument.
- `SubstringFunctionName`: Name of the SQL function for getting the substring of a string.
- `DistinctCountColumnSeparator`: The string the database uses to delimit between column expressions in a SELECT COUNT(DISTINCT column-list) clause. Defaults to `null` for most databases, meaning that multiple columns in a dis-

inct COUNT clause are not supported.

- `SupportsSelectForUpdate`: If true, then the database supports SELECT statements with a pessimistic locking clause. Defaults to true.
- `SupportsLockingWithDistinctClause`: If true, then the database supports FOR UPDATE select clauses with DISTINCT clauses.
- `SupportsLockingWithOuterJoin`: If true, then the database supports FOR UPDATE select clauses with outer join queries.
- `SupportsLockingWithInnerJoin`: If true, then the database supports FOR UPDATE select clauses with inner join queries.
- `SupportsLockingWithMultipleTables`: If true, then the database supports FOR UPDATE select clauses that select from multiple tables.
- `SupportsLockingWithOrderClause`: If true, then the database supports FOR UPDATE select clauses with ORDER BY clauses.
- `SupportsLockingWithSelectRange`: If true, then the database supports FOR UPDATE select clauses with queries that select a range of data using LIMIT, TOP or the database equivalent. Defaults to true.
- `SimulateLocking`: Some databases do not support pessimistic locking, which will result in a `JDOException` when a pessimistic transaction is attempted. Setting this property to true will bypass the locking check to allow pessimistic transactions even on databases that do not support locking. Defaults to false.
- `SupportsQueryTimeout`: If true, then the JDBC driver supports calls to `java.sql.Statement.setQueryTimeout`.
- `SupportsHaving`: Whether this database supports HAVING clauses in selects.
- `SupportsSelectStartIndex`: Whether this database can create a select that skips the first N results.
- `SupportsSelectEndIndex`: Whether this database can create a select that is limited to the first N results.
- `SupportsSubselect`: Whether this database supports subselects in queries.
- `RequiresAliasForSubselect`: If true, then the database requires that subselects in a FROM clause be assigned an alias.
- `BatchLimit`: The maximum number of SQL update statements to batch together. Set to 0 to disable SQL batching, or -1 for no limit.
- `BatchParameterLimit`: The maximum number of total parameters that can be batched together for a single batch update. Some databases can only handle a certain total number of prepared statement parameters in a single batch. This value will cause Kodo to flush a SQL batch once the number of batched statements times the number of bound parameters per statement exceeds this value. Set to 0 to disable SQL batching, or -1 for no limit.
- `SupportsUpdateCountsForBatch`: Whether the JDBC driver correctly returns the set of update counts when a batch statement is executed.
- `SupportsTotalCountsForBatch`: If a JDBC driver doesn't support batch update counts, whether it at least returns the total number of updates made when a batch statement is executed.
- `SupportsMultipleNontransactionalResultSets`: If true, then a nontransactional connection is capable of having multiple open `ResultSet` instances.
- `StorageLimitationsFatal`: If true, then any data truncation/rounding that is performed by the dictionary in order to store a value in the database will be treated as a fatal error, rather than just issuing a warning.

- `StoreLargeNumbersAsStrings`: Many databases have limitations on the number of digits that can be stored in a numeric field (for example, Oracle can only store 38 digits). For applications that operate 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. Defaults to `false`.
- `StoreCharsAsNumbers`: Set this property to `false` to store Java `char` fields as `CHAR` values rather than numbers. Defaults to `true`.
- `UseGetBytesForBlobs`: If `true`, then `ResultSet.getBytes` will be used to obtain blob data rather than `ResultSet.getBinaryStream`.
- `UseGetObjectForBlobs`: If `true`, then `ResultSet.getObject` will be used to obtain blob data rather than `ResultSet.getBinaryStream`.
- `UseSetBytesForBlobs`: If `true`, then `PreparedStatement.setBytes` will be used to set blob data, rather than `PreparedStatement.setBinaryStream`.
- `UseGetStringForClobs`: If `true`, then `ResultSet.getString` will be used to obtain clob data rather than `ResultSet.getCharacterStream`.
- `UseSetStringForClobs`: If `true`, then `PreparedStatement.setString` will be used to set clob data, rather than `PreparedStatement.setCharacterStream`.
- `ArrayTypeNames`: The overridden default column type for `java.sql.Types.ARRAY`. This is only used when the schema is generated by the mappingtool.
- `BigIntTypeName`: The overridden default column type for `java.sql.Types.BIGINT`. This is only used when the schema is generated by the mappingtool.
- `BinaryTypeName`: The overridden default column type for `java.sql.Types.BINARY`. This is only used when the schema is generated by the mappingtool.
- `BitTypeName`: The overridden default column type for `java.sql.Types.BIT`. This is only used when the schema is generated by the mappingtool.
- `BlobTypeName`: The overridden default column type for `java.sql.Types.BLOB`. This is only used when the schema is generated by the mappingtool.
- `CharTypeName`: The overridden default column type for `java.sql.Types.CHAR`. This is only used when the schema is generated by the mappingtool.
- `ClobTypeName`: The overridden default column type for `java.sql.Types.CLOB`. This is only used when the schema is generated by the mappingtool.
- `DateTypeName`: The overridden default column type for `java.sql.Types.DATE`. This is only used when the schema is generated by the mappingtool.
- `DecimalTypeName`: The overridden default column type for `java.sql.Types.DECIMAL`. This is only used when the schema is generated by the mappingtool.
- `DistinctTypeName`: The overridden default column type for `java.sql.Types.DISTINCT`. This is only used when the schema is generated by the mappingtool.
- `DoubleTypeName`: The overridden default column type for `java.sql.Types.DOUBLE`. This is only used when the schema is generated by the mappingtool.
- `FloatTypeName`: The overridden default column type for `java.sql.Types.FLOAT`. This is only used when the schema is generated by the mappingtool.
- `IntegerTypeName`: The overridden default column type for `java.sql.Types.INTEGER`. This is only used when the

schema is generated by the mappingtool.

- `JavaObjectTypeName`: The overridden default column type for `java.sql.Types.JAVA_OBJECT`. This is only used when the schema is generated by the mappingtool.
- `LongVarbinaryTypeName`: The overridden default column type for `java.sql.Types.LONGVARBINARY`. This is only used when the schema is generated by the mappingtool.
- `LongVarcharTypeName`: The overridden default column type for `java.sql.Types.LONGVARCHAR`. This is only used when the schema is generated by the mappingtool.
- `NullTypeName`: The overridden default column type for `java.sql.Types.NULL`. This is only used when the schema is generated by the mappingtool.
- `NumericTypeName`: The overridden default column type for `java.sql.Types.NUMERIC`. This is only used when the schema is generated by the mappingtool.
- `OtherTypeName`: The overridden default column type for `java.sql.Types.OTHER`. This is only used when the schema is generated by the mappingtool.
- `RealTypeName`: The overridden default column type for `java.sql.Types.REAL`. This is only used when the schema is generated by the mappingtool.
- `RefTypeName`: The overridden default column type for `java.sql.Types.REF`. This is only used when the schema is generated by the mappingtool.
- `SmallintTypeName`: The overridden default column type for `java.sql.Types.SMALLINT`. This is only used when the schema is generated by the mappingtool.
- `StructTypeName`: The overridden default column type for `java.sql.Types.STRUCT`. This is only used when the schema is generated by the mappingtool.
- `TimeTypeName`: The overridden default column type for `java.sql.Types.TIME`. This is only used when the schema is generated by the mappingtool.
- `TimestampTypeName`: The overridden default column type for `java.sql.Types.TIMESTAMP`. This is only used when the schema is generated by the mappingtool.
- `TinyintTypeName`: The overridden default column type for `java.sql.Types.TINYINT`. This is only used when the schema is generated by the mappingtool.
- `VarbinaryTypeName`: The overridden default column type for `java.sql.Types.VARBINARY`. This is only used when the schema is generated by the mappingtool.
- `VarcharTypeName`: The overridden default column type for `java.sql.Types.VARCHAR`. This is only used when the schema is generated by the mappingtool.
- `UseSchemaName`: If `false`, then avoid including the schema name in table name references. Defaults to `true`.
- `DefaultSchemaName`: The default schema name to use for schema interrogation. Defaults to `null`, which will check all schemas.
- `TableTypes`: Comma-separated list of table types to use when looking for tables during schema reflection, as defined in the `java.sql.DatabaseMetaData.getTableInfo` JDBC method. An example is: `"TABLE, VIEW, ALIAS"`. Defaults to `"TABLE"`.
- `SupportsSchemaForGetTables`: If `false`, then the database driver does not support using the schema name for schema reflection on table names.
- `SupportsSchemaForGetColumns`: If `false`, then the database driver does not support using the schema name for

schema reflection on column names.

- `SupportsNullTableForGetColumns`: If true, then the database supports passing a null parameter to `DatabaseMetaData.getColumns` as an optimization to get information about all the tables. Defaults to true.
- `SupportsNullTableForGetPrimaryKeys`: If true, then the database supports passing a null parameter to `DatabaseMetaData.getPrimaryKeys` as an optimization to get information about all the tables. Defaults to false.
- `SupportsNullTableForGetIndexInfo`: If true, then the database supports passing a null parameter to `DatabaseMetaData.getIndexInfo` as an optimization to get information about all the tables. Defaults to false.
- `SupportsNullTableForGetImportedKeys`: If true, then the database supports passing a null parameter to `DatabaseMetaData.getImportedKeys` as an optimization to get information about all the tables. Defaults to false.
- `UseGetBestRowIdentifierForPrimaryKeys`: If true, then metadata queries will use `DatabaseMetaData.getBestRowIdentifier` to obtain information about primary keys, rather than `DatabaseMetaData.getPrimaryKeys`.
- `RequiresAutoCommitForMetadata`: If true, then the JDBC driver requires that autocommit be enabled before any schema interrogation operations can take place.
- `AutoIncrementClause`: The column definition clause to append to a creation statement. For example, "AUTO\_INCREMENT" for MySQL. This property is set automatically in the dictionary, and should not need to be overridden, and is only used when the schema is generated using the `mappingtool`.
- `AutoIncrementTypeName`: The column type name for auto-increment columns. For example, "SERIAL" for PostgreSQL. This property is set automatically in the dictionary, and should not need to be overridden, and is only used when the schema is generated using the `mappingtool`.
- `LastGeneratedKeyQuery`: The query to issue to obtain the last automatically generated key for an auto-increment column. For example, "select @@identity" for Sybase. This property is set automatically in the dictionary, and should not need to be overridden.

### 4.3.1. MySQLDictionary parameters

---

The `mysql` dictionary also understands the following properties:

- `DriverDeserializesBlobs`: Many MySQL drivers automatically deserialize BLOBs on calls to `ResultSet.getObject`. The `MySQLDictionary` overrides the standard `DBDictionary.getBlobObject` method to take this into account. If your driver does not deserialize automatically, set this property to false.
- `TableType`: The MySQL table type to use when creating tables. Defaults to `innodb`.
- `UseClobs`: Some older versions of MySQL do not handle clobs correctly. To enable clob functionality, set this to true. Defaults to false.

### 4.3.2. OracleDictionary parameters

---

The `oracle` dictionary understands the following properties:

- `UseTriggersForAutoIncrement`: If true, then Kodo will allow simulation of auto-increment columns by the use of Oracle triggers. Kodo will assume that the current sequence value from the sequence specified in the `AutoIncrement-SequenceName` parameter will hold the value of the new primary key for rows that have been inserted. For more details on auto-increment support, see [Section 5.2.4.2, “Auto-Increment” \[187\]](#)

- `AutoIncrementSequenceName`: The global name of the sequence that Kodo will assume to hold the value of primary key value for rows that use auto-increment. If left unset, Kodo will use a the sequence named "SEQ\_<table name>".
- `MaxEmbeddedBlobSize`: Oracle is unable to persist Blobs using the embedded update method when Blobs get over a certain size. The size depends on database configuration, e.g. encoding. This property defines the maximum size Blob to persist with the embedded method. Defaults to 4000 bytes.
- `MaxEmbeddedClobSize`: Oracle is unable to persist Clobs using the embedded update method when Clobs get over a certain size. The size depends on database configuration, e.g. encoding. This property defines the maximum size Clob to persist with the embedded method. Defaults to 4000 characters.

## 4.4. Configuring the DBDictionary

---

The example below demonstrates how to set a dictionary and configure its properties in your configuration file. This property uses Kodo's **plugin syntax**.

### *Example 4.3. Specifying a DBDictionary*

```
kodo.jdbc.DBDictionary: hsql(SimulateLocking=true)
```

## 4.5. Accessing Multiple Databases

---

Through the properties we've covered thus far, 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 give the correct file to Kodo JDO's command-line tools during development.

## 4.6. Setting the Transaction Isolation

---

By default, Kodo JDO relies on the default transaction isolation level of the JDBC driver. However, you can specify a transaction isolation level to use through the `kodo.jdbc.TransactionIsolation` configuration property. The following is a list of standard isolation levels. Note that not all databases support all isolation levels.

- `default`: Use the JDBC driver's default isolation level. Kodo uses this option if you do not explicitly specify any other.
- `none`: No transaction isolation.
- `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.

### *Example 4.4. Specifying a Transaction Isolation*

```
kodo.jdbc.TransactionIsolation: repeatable-read
```

## 4.7. Setting the SQL Join Syntax

JDO queries often involve using SQL joins behind the scenes. You can configure Kodo to use either SQL 92-style join syntax, in which joins are placed in the SQL FROM clause, the traditional join syntax, in which join criteria are part of the WHERE clause, or a database-specific join syntax mandated by the **DBDictionary**. Kodo only supports outer joins when using SQL 92 syntax or a database-specific syntax with outer join support.

The **kodo.jdbc.DBDictionary** plugin accepts the **JoinSyntax** property to set the system's default syntax. Syntax can be changed on a per persistence manager, query, or extent basis using the **fetch configuration** API, which is described in the **JDO Runtime Extensions** chapter.

### *Example 4.5. Specifying the Join Syntax Default*

```
kodo.jdbc.DBDictionary: JoinSyntax=sql92
```

### *Example 4.6. Specifying the Join Syntax at Runtime*

```
import kodo.query.*;           // for KodoQuery
import kodo.jdbc.runtime.*;    // for JDBCFetchConfiguration

...

KodoQuery kq = (KodoQuery) pm.newQuery (MyClass.class, "foo == bar");
JDBCFetchConfiguration fetch = (JDBCFetchConfiguration)
    kq.getFetchConfiguration ();
fetch.setJoinSyntax (fetch.JOIN_SYNTAX_SQL92);
Collection results = (Collection) kq.execute ();
```

## 4.8. Configuring the Use of JDBC Connections

In its default configuration, Kodo JDO obtains JDBC connections on an as-needed basis. Kodo persistence managers do not retain a connection to the database unless they are in a datastore transaction or there are open extent iterators or query results that are using a live JDBC result set. At all other times, including during optimistic transactions, persistence managers request a connection for each query, then immediately release the connection back to the pool.

In some cases, it may be more efficient to retain connections for longer periods of time. You can configure Kodo JDO's use of JDBC connections through the **kodo.ConnectionRetainMode** configuration property. The property accepts the following values:

- **persistence-manager**: Each persistence manager obtains a single connection and uses it until the persistence manager

is closed.

- **transaction**: A connection is obtained when each transaction begins (optimistic or datastore), and is released when the transaction completes. Non-transactional connections are obtained on-demand.
- **on-demand**: Connections are obtained only when needed. This option is equivalent to the **transaction** option when datastore transactions are used. For optimistic transactions, though, it means that a connection will be retained only for the duration of the data store flush and commit process.

You can also specify the connection retain mode of individual persistence managers when you retrieve them from the persistence manager factory. See [Section 10.1, “KodoPersistenceManagerFactory” \[288\]](#) for details.

The **kodo.FlushBeforeQueries** configuration property controls another aspect of connection usage: whether to flush transactional changes before executing JDO queries. This setting only applies to queries that would otherwise have to be executed in-memory because the **IgnoreCache** property is set to false and the query may involve objects that have been changed in the current transaction. Legal values are:

- **true**: Always flush rather than executing the query in-memory. If the current transaction is optimistic, Kodo will begin a non-locking datastore transaction.
- **false**: Never flush before a query.
- **with-connection**: Flush only if the persistence manager has already established a dedicated connection to the data store, otherwise execute the query in-memory. This option is useful if you use long-running optimistic transactions and want to ensure that these transactions do not consume database resources until commit. Kodo's behavior with this option is dependent on the transaction status and mode, as well as the configured connection retain mode described earlier in this section.

The flush mode can also be set on individual Kodo JDO persistence manager and query instances using the **fetch configuration** API, discussed in the **JDO Runtime Extensions** chapter.

The table below describes 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 4.2. Kodo Automatic Flush Behavior**

	<b>FlushBeforeQueries = false</b>	<b>FlushBeforeQueries = true</b>	<b>FlushBeforeQueries = with-connection; ConnectionRetainMode = on-demand</b>	<b>FlushBeforeQueries = with-connection; ConnectionRetainMode = transaction or persistence-manager</b>
<b>IgnoreCache = true</b>	no flush	no flush	no flush	no flush
<b>IgnoreCache = false; no tx active</b>	no flush	no flush	no flush	no flush
<b>IgnoreCache = false; datastore tx active</b>	no flush	flush	flush	flush
<b>IgnoreCache = false; optimistic tx active</b>	no flush	flush	no flush unless KodoPersistenceManager.flush has already been invoked	flush

*Example 4.7. Specifying Connection Usage Defaults*

```
kodo.ConnectionRetainMode: on-demand
kodo.FlushBeforeQueries: true
```

*Example 4.8. Specifying Connection Usage at Runtime*

```
import kodo.runtime.*;

...

// obtaining a pm with a certain transaction and connection retain mode
KodoPersistenceManagerFactory pmf = (KodoPersistenceManagerFactory) JDOHelper.
    getPersistenceManagerFactory (props);
PersistenceManager pm = pmf.getPersistenceManager
    (KodoPersistenceManager.TRANS_LOCAL, KodoPersistenceManager.CONN_RETAIN_PM);

...

// changing the flush mode for an individual persistence manager
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
FetchConfiguration fetch = kpm.getFetchConfiguration ();
kpm.getFetchConfiguration ().setFlushBeforeQueries (fetch.QUERY_FLUSH_TRUE);
```

## 4.9. Runtime Access to JDBC Connections

Kodo JDO provides two mechanisms for obtaining a `java.sql.Connection` object. Accessing a connection can be useful when you require direct access to the underlying data store.

The following code obtains the connection that is currently in use by a particular persistence manager. If there is no connection open to the data store for the persistence manager, then a new one is created and returned. If a data store transaction is in progress, then the connection returned will be transactionally consistent.

### Note

Whether or not a persistence manager already has a connection open at any point in time is determined by whether a transaction is in progress, the type of the transaction (optimistic or datastore), and the setting of the `kodo.ConnectionRetainMode` property. It can also be influenced by whether or not you have explicitly flushed transactional changes, and the value of the `kodo.FlushBeforeQueries` property.

*Example 4.9. Obtaining a JDBC Connection from the PersistenceManager*

```
import java.sql.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
Connection conn = (Connection) kpm.getConnection ();

// do stuff
```

```
conn.close ();
```

The connection should be closed regardless of the current transactional state, and regardless of whether or not the persistence manager is being closed. The connection returned from the `KodoPersistenceManager` will ignore `close` invocations as needed to maintain transactional integrity.

Additionally, you can request a connection that is independent of the current persistence manager:

### *Example 4.10. Obtaining a JDBC Connection from the DataSource*

```
import java.sql.*;
import javax.sql.*;
import kodo.conf.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
JDOConfiguration conf = kpm.getConfiguration ();
DataSource dataSource = (DataSource) conf.getConnectionFactory ();
Connection conn = dataSource.getConnection ();

// do stuff

conn.close ();
```

## 4.10. Large Result Sets

---

By default, Kodo uses standard forward-only JDBC result sets, and completely instantiates the results of queries on execution. When using a JDBC driver that supports version 2.0 or higher of the JDBC specification, however, you can configure Kodo to use scrolling result sets that may not bring all results into memory at once. You can also configure the number of result objects Kodo keeps references to, so that you can traverse potentially enormous amounts of data without exhausting JVM memory. On-demand loading is also applied to extent iterators, and can be configured for individual collection and map fields via **large result set proxies**.

To configure Kodo's handling of result sets, use the following properties:

- **kodo.FetchBatchSize**: The number of objects to instantiate at once when traversing a result set. This number will be set as the fetch size on JDBC Statement objects used to obtain result sets. It also factors in to the number of objects Kodo will maintain a hard reference to when traversing a query result.

The fetch size defaults to -1, meaning all results will be instantiated immediately on query execution. A value of 0 means to use the JDBC driver's default batch size. Thus to enable large result set handling, you must set this property to 0 or to a positive number.

- **kodo.jdbc.ResultSetType**: The type of result set to use when executing queries and traversing extents. This property accepts the following values, each of which corresponds exactly to the same-named `java.sql.ResultSet` constant:
  - `forward-only`: This is the default.
  - `scroll-sensitive`

- `scroll-insensitive`

Different JDBC drivers treat the different result set types differently. Also, not all drivers support all types.

- **`kodo.jdbc.FetchDirection`**: The expected order in which you will access the query results. This property affects the type of datastructure Kodo will use to hold the results, and is also given to the JDBC driver in case it can optimize for certain access patterns. This property accepts the following values, each of which corresponds exactly to the same-named `java.sql.ResultSet` `FETCH` constant:

- `forward`: This is the default.
- `reverse`
- `unknown`

Not all drivers support all fetch directions.

- **`kodo.jdbc.LRSSize`**: The strategy Kodo will use to determine the size of result sets. This property is **only** used if you change the fetch batch size from its default of -1, so that Kodo begins to use on-demand result loading. Available values are:
  - `query`: This is the default. The first time you ask for the size of a query result, Kodo will perform a `SELECT COUNT( * )` query to determine the number of expected results. Note that depending on transaction status and settings, this can mean that the reported size is slightly different than the actual number of results available.
  - `last`: If you have chosen a scrollable result set type, this setting will use the `ResultSet.last` method to move to the last element in the result set and get its index. Unfortunately, some JDBC drivers will bring all results into memory in order to access the last one. Note that if you do not choose a scrollable result set type, then this will behave exactly like `unknown`. The default result set type is `forward-only`, so you must change the result set type in order for this property to have an effect.
  - `unknown`: Under this setting Kodo will return `Integer.MAX_VALUE` as the size for any query result that uses on-demand loading. This is allowed by the JDO specification, but clearly isn't helpful if you actually need the number of returned results.

### *Example 4.11. Specifying Result Set Defaults*

```
kodo.FetchBatchSize: 20
kodo.jdbc.ResultSetType: scroll-insensitive
kodo.jdbc.FetchDirection: forward
kodo.jdbc.LRSSize: last
```

Many **Kodo JDO runtime components** such as the `KodoPersistenceManager`, `KodoQuery`, and `KodoExtent` also have methods to configure these properties on a case-by-case basis through their **fetch configuration** object.

### *Example 4.12. Specifying Result Set Behavior at Runtime*

```
import java.sql.*;
import kodo.query.*;
import kodo.jdbc.runtime.*;

...

KodoQuery kq = (KodoQuery) pm.newQuery (MyClass.class, "foo == bar");
JDBCFetchConfiguration fetch = (JDBCFetchConfiguration)
    kq.getFetchConfiguration ();
fetch.setFetchBatchSize (20);
fetch.setResultSetType (ResultSet.TYPE_SCROLL_INSENSITIVE);
fetch.setFetchDirection (ResultSet.FETCH_FORWARD);
fetch.setLRSSize (JDBCFetchConfiguration.SIZE_LAST);
Collection results = (Collection) kq.execute ();
```

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 4.13. Using Random Access Query Results in a Portable Fashion*

```
// we want to ensure that we always order the results in the same way
Query query = pm.newQuery (Product.class, "productName == 'Stereo'");
query.setOrdering ("productCode ascending");

Collection results = (Collection) query.execute ();
List resultList;
if (results instanceof List) // always the case with Kodo JDO
    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);

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

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

query.close (results);
```

## 4.11. SQL Statement Ordering & Foreign Keys

Kodo JDO can be configured to order SQL updates to meet foreign key constraints through the boolean `kodo.jdbc.ForeignKeyConstraints` configuration property.

SQL statement ordering to match foreign key constraints is disabled by default. If you use foreign keys in your schema, we recommend configuring your database to use deferred foreign keys whenever possible. Deferred constraints are not evaluated until the end of the transaction, giving you greater freedom in how you design your constraints and better performance than Kodo JDO's in-memory statement ordering. If your database does not support deferred constraints, however, Kodo JDO's automatic statement ordering will make sure all dependencies are met, including circular dependencies. The example below shows how to enable statement ordering in your configuration properties file.

### *Example 4.14. Enabling SQL Statement Ordering*

```
kodo.jdbc.ForeignKeyConstraints: true
```

If you are using Kodo to create your schema, you should consult [Section 6.2.2.14, “jdbc-delete-action” \[202\]](#) for information on how to tell Kodo to create foreign key constraints at schema generation time.

### Note

You cannot use non-deferred foreign key constraints with the dynamic schema factory. Please make sure you are using one of the other factories covered in [Section 8.1.2, “Schema Factory” \[278\]](#)

---

# Chapter 5. Persistent Classes

Persistent class basics are covered in [Chapter 4, \*PersistenceCapable\* \[16\]](#) of the JDO Overview. This chapter details the tools Kodo JDO provides to aid in persistent class creation.

## 5.1. Restrictions on Persistent Classes

---

Kodo JDO places very few restrictions on persistent classes, other than those mandated by the JDO specification, which we have enumerated in [Section 4.3, “Restrictions on Persistent Classes” \[17\]](#). In particular, Kodo JDO fully supports inheritance, as well as a wide array of persistent field types and persistent relations between objects. See [Chapter 7, \*Object-Relational Mapping\* \[209\]](#) for a complete list of the supported inheritance and persistent field patterns.

## 5.2. Object Identity

---

Kodo supports both datastore and application JDO identity types, including JDO 2's single field application identity (see [Section 4.5, “JDO Identity” \[21\]](#) in the JDO Overview for a refresher on JDO identity types).

### 5.2.1. Datastore Identity

---

For datastore identity, Kodo uses the public `kodo.util.Id` class. You can manipulate datastore oid values returned by Kodo JDO by casting them to this class. You can also create your own `Id` instances and pass them to any Kodo JDO method that expects a datastore oid object. Remember, however, that datastore identity in JDO is meant to be opaque; if you find yourself having to use the `Id` API often, it may be a sign that you should be using application identity.

### 5.2.2. Application Identity

---

If you choose to use application identity, you may want to take advantage of Kodo JDO's application identity tool. The application identity tool 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 application identity tool on a persistent class, the class must be compiled and must have complete JDO metadata. Set the metadata's `objectId-class` attribute to the desired name of the generated object identity class, and be sure to include `<field>` elements for all primary key fields, with the `primary-key` attribute set to `true`.

The application identity tool can be invoked via the included `appidtool` shell/bat script or via its Java class, `kodo.enhance.ApplicationIdTool`.

#### *Example 5.1. Using the Application Identity Tool*

```
appidtool package.jdo
```

The application identity tool accepts the standard set of command-line arguments defined by the configuration framework (see [Section 2.3, “Command Line Configuration” \[139\]](#)), including code formatting flags described in [Section 2.3.1, “Code Formatting” \[140\]](#). It also accepts the following arguments:

- `-token/-t <token>`: The token to use to separate stringified primary key values in the string form of the object id. This

option is only used if you have multiple primary key fields. It defaults to ":".

- `-ignoreErrors/-i <true/t | false/f>` : If `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 (recall that subclasses never define the application identity class; they inherit it from their persistent superclass).
- `-directory/-d <output directory>`: Path to the output directory. If the directory does not match the generated oid class' package, the package structure will be created beneath the directory. If not specified, the tool will first try to find the directory of the `.java` file for the persistence-capable class, and failing that will use the current directory.

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

- The full name of a persistent class.
- The `.java` file for a persistent class.
- The `.class` file of a persistent class.
- A `.jdo` metadata file. The tool will run on each class listed in the metadata.

### Note

When running Kodo JDO tools on `.java` files, only the top-level class for the file will be processed. Inner classes are ignored.

---

## 5.2.3. Single Field Identity

The JDO Overview describes single field identity in [Section 4.5.3, “Single Field Identity” \[26\]](#). Pay particular attention to the note in this section detailing Kodo's treatment of single field identity as a JDO 2 preview feature. Also, note that until JDO 2 jars are available, using the single field identity preview will introduce Kodo dependencies into your persistent classes during enhancement.

---

## 5.2.4. Primary Key Generation

Kodo supports three styles of primary key generation: sequence factories, sequence assigned fields, and auto-increment columns.

---

### 5.2.4.1. Sequence Factory

To generate unique datastore identity primary key values, Kodo JDO typically uses a `kodo.jdbc.schema.SequenceFactory` internally. The default sequence factory implementation in use is controlled by the `kodo.jdbc.SequenceFactory` configuration property. You can also declare that a given class uses a specific sequence factory through the `jdbc-sequence-factory` metadata extension.

Using a sequence factory allows coarse-grained control over primary key values while maintaining the simplicity of using datastore identity. Furthermore, it is possible to use the sequence factory yourself to make default assignments to the fields of application identity instances. This gives you the fine-grained control of application identity, without the burden of coming up with your own system to generate unique primary key values. Kodo can also automatically assign values from this sequence factory. See [Section 5.2.4.3, “Sequence-Assigned” \[188\]](#)

Kodo JDO ships with four available sequence factories, and you are free to substitute your own as well.

- `db`: This is the default. It is an alias for the `kodo.jdbc.schema.DBSequenceFactory` class. The `DBSequence-`

Factory uses a special single-row table to store a global sequence number. If the table does not already exist, the factory will create it the first time you run the **mapping tool**'s refresh action. The DBSequenceFactory accepts the following properties:

- **TableName**: The name of the special sequence number table to use. Defaults to JDO\_SEQUENCE.
- **PrimaryKeyColumn**: The name of the primary key column for the sequence table. Defaults to ID.
- **SequenceColumn**: The name of the column that will hold the current sequence value. Defaults to SEQUENCE\_VALUE.
- **Increment**: The amount to increment the sequence number by. Defaults to 50, meaning the factory will set aside the next 50 numbers each time it accesses the sequence table, which in turn means it only has to make a database trip to get new sequence numbers once every 50 object inserts.

Using the default settings, the DDL for the sequence table will look like this:

```
CREATE TABLE JDO_SEQUENCE(ID TINYINT NOT NULL PRIMARY KEY, SEQUENCE_VALUE BIGINT)
```

- **db-class**: This is an alias for the **kodo.jdbc.schema.ClassDBSequenceFactory**, which extends the DBSequenceFactory above with the ability to maintain a separate sequence number per-class. It shares the same properties as the DBSequenceFactory. Instead of maintaining a global sequence number in a single row, though, this factory's sequence table contains a sequence number for each persistent class, one per row.
- **native**: This is an alias for the **kodo.jdbc.schema.ClassSequenceFactory**. Many databases have a concept of "native sequences" -- a built-in mechanism for obtaining monotonically incrementing numbers. For example, in Oracle, a database sequence can be created with a statement like `CREATE SEQUENCE MYSEQUENCE`. Sequence values can then be atomically obtained and incremented with the statement `SELECT MYSEQUENCE.NEXTVAL FROM DUAL`. Kodo JDO provides support for this common mechanism of primary key generation with the ClassSequenceFactory. The factory accepts the following properties:
  - **TableName**: The table name to run sequence queries against. Defaults to DUAL.
  - **Format**: The string used to generate the SQL for selecting a sequence value. The string can have two placeholders: {0} for the sequence name and {1} for the table name. The default is `SELECT {0}.NEXTVAL FROM {1}`.
  - **SequenceName**: The name of the default sequence. Defaults to JDOSEQUENCE. Each class can also declare what sequence name to use for members of that class through the `jdbc-sequence-name` class-level **metadata extension**.
- **sjvm**: This is an alias for the **kodo.jdbc.schema.SimpleSequenceFactory**. This factory uses an in-memory static counter, initialized to the current time in milliseconds and monotonically incremented for each new object. It is only suitable for single-JVM environments.

### Example 5.2. Sequence Factory Configuration

A hypothetical excerpt from `kodo.properties` to use the ClassSequenceFactory.

```
kodo.jdbc.SequenceFactory: native(TableName=SEQTABLE, \
  Format="SELECT {0}.NEXT FROM {1}")
```

The assigned sequence factory for the class/configuration can be used to assign field values on flush. This is especially useful for generating primary key values in application identity classes. See [Section 5.2.4.3, “Sequence-Assigned” \[188\]](#)

As mentioned above, you may want to use the sequence factory in your application. The example below demonstrates how to do so.

### ***Example 5.3. Accessing the Sequence Factory***

You may find it useful to access the sequence factory yourself to help create unique application identity field values. You can accomplish this by getting a `SequenceGenerator` instance, which is a delegate to the `SequenceFactory`:

```
import kodo.runtime.*;

...

SequenceGenerator gen = KodoHelper.getSequenceGenerator (pm, MyPCClass.class);
pm.currentTransaction ().begin ();

MyPCClass pc1 = new MyPCClass ();
pc1.setPrimaryKey (gen.getNext ().longValue ());
pm.makePersistent (pc1);

MyPCClass pc2 = new MyPCClass ();
pc2.setPrimaryKey (gen.getNext ().longValue ());
pm.makePersistent (pc2);

pm.currentTransaction ().commit ();
```

---

## **5.2.4.2. Auto-Increment**

Some databases allow you to define columns that are automatically assigned a unique numeric value when a record is inserted. In many databases, this value is monotonically increasing. Thus, Kodo JDO refers to these columns as *auto-increment* columns.

Any field can use an auto-increment column in Kodo, but they are most often used for primary key fields under application identity, or for the primary key column under datastore identity. Before you decide to use auto-increment columns, you should be aware of the following restrictions:

1. Auto-increment columns must be an integer or long integer type.
2. Databases support auto-increment columns to varying degrees. Some do not support them at all. Others only allow a single auto-increment column per table, and require that it be the primary key column. More lenient databases may allow non-primary key auto-increment columns, and may allow more than one per table. See your database documentation for details.
3. Statements inserting into tables with auto-increment columns cannot be batched. After each insert, Kodo must go back to the database to retrieve the last inserted auto-increment value to set back in the persistent object. This can have a negative impact on performance.
4. Requesting the object id of persistent-new objects using auto-increment columns for their primary keys will cause a flush so that the id can be determined.

The **`jdbc-auto-increment`** metadata extension controls Kodo's use of auto-increment columns. Placing this extension beneath the `<class>` element indicates that the datastore identity primary key column of the class will auto-increment. Placing it on a `<field>` element means that the field's column will auto-increment. You must specify these extensions even if you do not use Kodo to create your schema, because Kodo cannot reliably determine which columns are auto-incrementing through the JDBC driver alone.

### *Example 5.4. Auto-Increment Metadata*

In the example below, class `Person` uses an auto-incrementing datastore identity primary key column, while class `Form` uses an auto-incrementing primary key field with application identity.

```
<jdo>
<package name="com.xyz">
  <class name="Person">
    <extension vendor-name="kodo" key="jdbc-auto-increment" value="true"/>
  </class>
  <class name="Form" objectid-class="FormId">
    <field name="id" primary-key="true">
      <extension vendor-name="kodo" key="jdbc-auto-increment" value="true"/>
    </field>
  </class>
</package>
</jdo>
```

---

#### 5.2.4.3. Sequence-Assigned

Sequence factories can be applied to classes using application identity as well. This is useful for generating unique primary key values automatically. These values will be generated when the owning instance is made persistent. Combined with **single field identity**, this feature simplifies the use of application identity. This field level extension can be used for any long compatible field (i.e. long, short, integer or their wrapper types).

Kodo will use the same **sequence factory** assigned to the class to generate the next value. If multiple fields are marked as sequence assigned, Kodo will assign unique values for each field. If a non-default value was already assigned when `PersistenceManager.makePersistent()` (or implied during persistence by reachability), the field will not be assigned a value.

### *Example 5.5. Sequence-Assigned Metadata*

In this example, class `Company` will have a single primary key field `id`. This field's value will be auto-assigned a value from the system sequence factory as the class does not declare a specific one.

```
<jdo>
<package name="com.xyz">
  <class name="Company" objectid-class="CompanyId">
    <field name="id" primary-key="true">
      <extension vendor-name="kodo" key="sequence-assigned" value="true"?>
    </field>
  </class>
</package>
</jdo>
```

---

## 5.3. Managed Inverses

Bidirectional relations are an essential part of data modeling. Java does not provide any native facilities to ensure that both sides of a bidirectional relation remain consistent. Whenever you set one side of the relation, you must manually set the other side as well.

By default, Kodo behaves the same way. Kodo does not automatically propagate changes from one field in bidirectional relation to the other field. This is in keeping with JDO's philosophy of transparency, and also provides higher performance, as Kodo does

not need to analyze your object graph to correct inconsistent relations.

If convenience is more important to you than strict transparency, however, you can enable inverse relation management in Kodo. Set the `kodo.InverseManager` plugin property to `true` for standard management. Under this setting, Kodo detects changes to either side of a bidirectional relation, and automatically sets the other side appropriately on flush.

### *Example 5.6. Enabling Managed Inverses*

```
kodo.InverseManager: true
```

The inverse manager has options to log a warning or throw an exception when it detects an inconsistent bidirectional relation, rather than correcting it. To use these modes, set the manager's `Action` property to `warn` or `exception`, respectively.

By default, Kodo excludes **large result set fields** from management. You can force large result set fields to be included with the `ManageLRS` plugin property, as in the example below.

### *Example 5.7. Log Inconsistencies, Including LRS Fields*

```
kodo.InverseManager: true(Action=warn, ManageLRS=true)
```

Use the `inverse-owner` or `inverse-logical` metadata extensions to name the inverse of a bidirectional relation. The former extension implies that the fields share the same datastructure in the database, while the latter does not. See [Section 6.2.1.1, “inverse-owner” \[198\]](#) and [Section 6.2.1.2, “inverse-logical” \[198\]](#) for details.

## 5.4. Mutable Second Class Object Fields

---

Mutable second class objects consist of collections, maps, and other persistent field values that can be modified directly. This section documents some aspects of Kodo's handling of these fields that may affect the way you design your persistent classes.

### 5.4.1. Restoring Mutable Fields

---

JDO requires that all immutable fields be restored to their pre-transaction state when a transaction rollback occurs. Kodo also has the ability to restore the state of mutable fields including collections, maps, and arrays. To have the state of mutable fields restored on rollback, set the `kodo.RestoreMutableValues` configuration property to `true`.

### 5.4.2. Typing and Ordering

---

Kodo attempts to preserve the Java type and order of all collections and maps. By default, Kodo uses an ordering column to maintain sequencing for all fields declared to be a `List` type. You can turn sequencing on or off for these and other fields through the `jdbc-ordered metadata extension`. When loading data into a field, Kodo also examines the value you assign the field in your declaration code or in your no-args constructor. If the field value's type is more specific than the field's declared type, Kodo uses the value type to hold the loaded data. Kodo also uses the comparator you've initialized the field with, if any. Therefore, you can use custom comparators on your persistent field simply by setting up the comparator and using it in your field's initial value.

### *Example 5.8. Using Initial Field Values*

```
public class Company
{
    // Kodo will detect the custom comparator in the initial field value
    // and use it whenever loading data from the database into this field
    private Collection employeesBySal = new TreeSet (new SalaryComparator ());
    private Map departments;

    public Company
    {
        // or we can initialize fields in our no-args constructor; even though
        // this field is declared type Map, Kodo will detect that it's actually
        // a TreeMap and use natural ordering for loaded data
        departments = new TreeMap ();
    }

    // rest of class definition...
}
```

## 5.4.3. Proxies

---

At runtime, the values of all mutable 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. Kodo extends this standard JDO proxying behavior with smart proxies and custom proxies.

### 5.4.3.1. Smart Proxies

---

Most proxies only track whether or not they have been modified. Smart proxies for collection and map fields, however, keep a record of which elements have been added, removed, and changed. 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. Smart proxies for these types are more efficient than proxies for `Lists`. 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.

### 5.4.3.2. Large Result Set Proxies

---

Under standard JDO behavior, traversing a persistent collection or map field brings the entire contents of that field into memory. Some persistent fields, however, might represent huge amounts of data, to the point that attempting to fully instantiate them can overwhelm the JVM or seriously degrade performance.

Kodo uses special proxy types to represent these "large result set" fields. Kodo's large result set proxies do not cache any data in memory. Instead, each operation on the proxy offloads the work to the database and returns the proper result. For example, the `contains` method of a large result set collection will perform a `SELECT COUNT( * )` query with the proper where conditions to find out if the given element exists in the database's record of the collection. Similarly, each time you obtain an iterator Kodo performs the proper query using the current persistence manager's **large result set settings**, as discussed in the **JDBC** chapter. As you invoke `Iterator.next`, Kodo instantiates the result objects on-demand. You can free the resources used by a large result set iterator by passing it to the static **KodoHelper.close** method.

### *Example 5.9. Using a Large Result Set Iterator*

```
import kodo.runtime.*;
```

```
...  
Collection employees = company.getEmployees (); // employees is a lrs collection  
Iterator itr = employees.iterator ();  
try  
{  
    while (itr.hasNext ())  
        process ((Employee) itr.next ());  
}  
finally  
{  
    KodoHelper.close (itr);  
}
```

You can also add and remove from large result set proxies, just as with standard fields. Kodo keeps a record of all changes to the elements of the proxy, which it uses to make sure the proper results are always returned from collection and map methods, and to update the field's database record on commit.

In order to use large result set proxies, you must mark each large result set field with the **lrs** metadata extension. The extension takes a value of `true` or `false`. The following restrictions apply to large result set fields:

- The field must be declared as either a `java.util.Collection` or `java.util.Map`. It cannot be declared as any other type, including any sub-interface of collection or map, or any concrete collection or map class.
- The field cannot have an externalizer (see [Section 7.9.23, “Externalization” \[273\]](#))
- Because they rely on their owning object for context, large result set proxies cannot be transferred from one persistent field to another. The following code would result in an error on commit:

```
Collection employees = company.getEmployees () // employees is a lrs collection  
company.setEmployees (null);  
anotherCompany.setEmployees (employees);
```

### *Example 5.10. Marking a Large Result Set Field*

```
<class name="Company">  
  <field name="employees">  
    <collection element-type="Employee"/>  
    <extension vendor-name="kodo" key="lrs" value="true"/>  
  </field>  
  ...  
</class>
```

---

## 5.4.3.3. Custom Proxies

In Kodo JDO, proxies are managed through the `kodo.util.ProxyManager` interface. Kodo JDO includes a default proxy manager, the `kodo.util.ProxyManagerImpl` (with a plugin alias name of `default`), that will meet the needs of most users. The default proxy manager understands the following configuration properties:

- `TrackChanges`: Whether to use **smart proxies**. Defaults to `true`.
- `AssertAllowedType`: Whether to check elements of maps and collections as they are added to make sure they are of the type defined as the `key-type`, `value-type`, or `element-type` defined in the JDO metadata. Defaults to `false`.

For custom behavior, Kodo JDO allows you to define your own proxy classes, and your own proxy manager. See the `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 `kodo.ProxyManager` configuration property.

Your Kodo JDO distribution includes custom proxy samples in the `samples/proxies` directory.

### *Example 5.11. Configuring the Proxy Manager*

```
kodo.ProxyManager: AssertAllowedType=true, TrackChanges=false
```

---

## 5.5. Enhancement

As discussed in the **JDO Overview**, JDO uses a process called *enhancement* to prepare persistent classes for management by the JDO runtime. The Kodo JDO enhancer is a command-line tool that can be invoked via the included `jdoc` script or via its Java class, `kodo.enhance.JDOEnhancer`.

### *Example 5.12. Using the Kodo JDO Enhancer*

```
jdoc package.jdo
```

The enhancer accepts the standard set of command-line arguments defined by the configuration framework (see **Section 2.3, “Command Line Configuration”** [139]), along with the following flags:

- `-directory/-d <output directory>`: Path to the output directory. If the directory does not match the enhanced class' package, the package structure will be created beneath the directory. By default, the enhancer overwrites the original `.class` file.

Like the application identity tool, each additional argument to the enhancer must be either the full name of a persistent class, the `.java` file 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 en-

hanced, class loading errors can occur. In the event of a class load error, simply re-compile and re-enhance the offending classes.

## 5.6. Auto-Generating Classes from a Schema

---

Kodo JDO includes a reverse mapping tool for generating persistent class definitions, complete with JDO metadata and object-relational mapping data, from an existing database schema. You do not have to use the reverse mapping tool to access an existing schema; you are free to write your classes and **mapping information** by hand. The reverse mapping tool, however, can give you an excellent starting point from which to grow your persistent classes.

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

1. Use the **schema generator** to export your current schema to an XML schema file. You can skip this step and the next step if you want to run the reverse mapping tool directly against the database schema.

### *Example 5.13. Using the Schema Generator*

```
schemagen -f schema.xml
```

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 for the schema file is described in **Section 8.3, “XML Schema Format” [282]**

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 schema in the database). The tool can be run via the included `reversemappingtool` script, or through its Java class, `kodo.jdbc.meta.ReverseMappingTool`.

### *Example 5.14. Using the Reverse Mapping Tool*

```
reversemappingtool -pkg com.xyz -d ~/src -cp customizer.properties schema.xml
```

In addition to the **standard configuration flags**, including **code formatting options**, the reverse mapping tool recognizes the following command line arguments:

- `-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. Each element of the list must follow the naming conventions for the `kodo.jdbc.Schemas` property. In fact, if this flag is omitted, it defaults to the value of the `Schemas` property. If the `Schemas` property is not defined, all schemas will be reverse-mapped.
- `-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.
- `-blobAsObject/-bo <true/t | false/f>` : By default, all binary columns are mapped to `byte[]` fields. Set this flag to `true` to map them to `Object` fields instead. Note that when mapped this way, the column is presumed to contain a serialized Java object.
- `-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.
- `-oneToManyRelations/-omr <true/t | false/f>`: Set to `false` to prevent the creation of inverse 1-many relations for every 1-1 relation detected.
- `-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.
- `-innerAppId/-inn <true/t | false/f>`: Set to `true` to have any generated application identity classes be created as static inner classes within the persistent classes. Defaults to `false`.
- `-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`.
- `-customizerClass/-cc <class name>` : The full class name of a **kodo.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 **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.

Running the tool will generate `.java` files for each generated class and its application identity class, package-level or per-class `.jdo` files, depending on the `-metadata` flag, and package-level or per-class `.mapping` files, depending on the same flag.

4. Examine the generated class, metadata, and mapping information, and modify it as necessary. Information on object-relational mapping in Kodo JDO is available in **Chapter 7, Object-Relational Mapping [209]** Remember that the reverse mapping tool only provides a starting point, and you are free to make whatever modifications you want to the code it generates.

After you are satisfied with the generated classes and their mappings, you should first compile them with `javac`, `jikes`,

or your favorite Java compiler. Make sure the classes and their JDO metadata are located in the directory corresponding to the `-package` flag you gave the reverse mapping tool.

Finally, use the **mapping tool** to import the object-relational mapping data in the mapping files into the **mapping factory** you have chosen to use. Note that if you are using the default **FileMappingFactory**, you can simply leave the mapping file(s) in place, without importing them.

### *Example 5.15. Using the Mapping Tool*

```
jikes *.java
mappingtool -a import xyz.mapping
```

Your persistent classes are now ready to access your existing schema. Remember to enhance them before use.

## 5.6.1. Customizing Reverse Mapping

---

The reverse mapping process can be customized with the `kodo.jdbc.meta.ReverseCustomizer` plugin interface. See the class **Javadoc** for details on the hooks that 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 `kodo.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 package. 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 package. 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.

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

### *Example 5.16. Customizing Reverse Mapping with Properties*

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

Example custom.properties:

```
com.xyz.TblMagazine.rename:      com.xyz.Magazine
com.xyz.TblArticle.rename:      com.xyz.Article
com.xyz.TblPubCompany.rename:   com.xyz.pub.Company
com.xyz.TblSysInfo.rename:      none

com.xyz.Magazine.allArticles.rename:  articles
com.xyz.Magazine.articles.type:      java.util.Collection
com.xyz.Magazine.articles.value:     new TreeSet()
com.xyz.Magazine.identity:           datastore

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

Your Kodo download includes the `PropertiesReverseCustomizer` source code. You can use this code as an example when writing your own customization class.

## 5.7. Persistent Class List

---

There is a known deficiency in the JDO specification whereby locating a persistent instance by application identity object is not possible until the class of the instance has been loaded by the JVM. This is typically not a problem, but in environments with non-standard class loading and under certain application designs, you may see odd behavior. To work around this deficiency, Kodo allows you to explicitly specify all of your persistent classes as a comma-separated list of full class names in the **kodo.PersistentClasses** configuration property. This list is entirely optional. All listed classes will be loaded into the JVM every time you obtain a new persistence manager.

This property is also used to optimize subclass identification: normally, the standard class indicator identifies subclasses by issuing a `SELECT DISTINCT <class indicator column>` query the first time a base class is used. If you specify this property, however, then the given list is used to calculate subclasses instead.

If this property is used at all, it must name every persistent class. Failure to do so will result in a warning being logged, and may disrupt the inheritance system.

---

# Chapter 6. Metadata

The **JDO Overview** covers JDO metadata basics. This chapter discusses the tools Kodo JDO provides to aid in metadata creation, and metadata extensions that Kodo JDO recognizes.

## 6.1. Generating Default JDO Metadata

---

Kodo JDO includes a metadata tool for generating default JDO metadata for your persistent classes. The tool can only rely on reflection, so 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 can be run via the included `metadatatool` shell/bat script, or through its Java class, `kodo.meta.JDOMetaDataTool`.

### *Example 6.1. Using the MetaDataTool*

```
metadatatool -f mypackage/package.jdo mypackage/*.java
```

In addition to the **standard configuration flags** accepted by all Kodo JDO tools, the metadata tool recognizes the following command line flags:

- `-verbose/-v <true/t | false/f>`: The metadata tool honors 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. If you set this flag to `true`, however, the tool will generate `<field>` elements for every persistent field.
- `-file/-f <metadata file>`: The name of the metadata file to generate. If this argument is not supplied, the tool will print the generated metadata to standard output.

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.

## 6.2. JDO Metadata Extensions

---

Kodo JDO takes advantage of JDO metadata's built-in extension mechanism to allow you to specify persistence-related information in the following categories:

- **Relations**: Relation extensions are used to declare related objects that should be automatically deleted when the parent object is deleted, to indicate the "owning" side of two-sided relations, and to give Kodo JDO more information on field types.
- **Schema**: You can use schema extensions to dictate column sizes, indexes, and foreign keys.
- **Object-Relational Mapping**: Using object-relational mapping extensions, you can customize how your object is mapped to the database.

- **Miscellaneous:** Other extensions, including custom fetch group configuration, caching hints, and sequence information.

All metadata extensions are optional; Kodo JDO will rely on its defaults when no explicit data is provided. 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.

## 6.2.1. Relation Extensions

---

One use of relation extensions is to indicate the "owning" side of two-sided relations. Kodo JDO never requires you to use two-sided relations, but doing so allows Kodo to create a more efficient schema by sharing tables and columns between both sides of the relation. For example, consider a hypothetical two-sided relation between the `Company` and `Employee` classes. Each `Employee` object stores a reference to a single `Company` in its `employer` field. Similarly, each `Company` tracks all of its `Employees` in an `employees` collection.

In this example, there is a logical relationship between the `Employee.employer` and `Company.employees` fields; the relationship is clearly two-sided. In fact, it is what is known as a *one-to-many* relationship. If we consider how this relationship could be efficiently modelled in the database, we see that each `Employee` record should store the primary key values of its `employer`. Each `Company` record, on the other hand, does not need any reference to its `employees`, because we can model the relation with a simple `SELECT`. To find all employees of company X, we `SELECT` all `Employee` records where the record's `employer` foreign key values matches company X's primary key values.

Because two-sided relations share data structures, Kodo JDO requires you to specify which side of the relation should be used to update the shared structures. This side of the relation is the "owner". In one-to-many relationships like the one between `Company` and `Employee`, the field holding a reference to the single object is always the owner (in this case `Employee.employer`). In other two-sided relations, the choice of which side owns the relation is arbitrary.

Another use of relation extensions is to mark a field as *dependent*. Objects stored in dependent fields are called dependent objects. Any time a dependent object is removed from its owning field or has its owning object deleted, the dependent object also becomes a candidate for deletion. On flush, Kodo checks whether you have added the dependent object to any other field in the transaction, and if you have not, the unreferenced object is deleted.

The final use of relation extensions is to give Kodo JDO extra information about field types. This is useful when you want your Java code to treat a field as a generic `java.lang.Object` or interface, but you know that the field will actually always hold a relation to another persistent object. Telling Kodo JDO that a generic object or interface field actually stores a persistent relation lets Kodo JDO create a more efficient schema.

### 6.2.1.1. inverse-owner

---

Use the `inverse-owner` field extension to indicate that this field is part of a two-sided relation with the named field, and that the named field owns the relation. The named field should **not** also have this extension; only one field can be the owner. See below for an example.

This extension implies that this field and its inverse owner share a datastructure at the database level. For a purely logical inverse, use **`inverse-logical`**.

By default, this extension does not affect runtime behavior. You must still set both sides of the relation manually to keep your object model consistent. You can, however, configure Kodo to manage the relation so that setting one side automatically sets the other side as well. See [Section 5.3, "Managed Inverses" \[188\]](#) for details.

### 6.2.1.2. inverse-logical

---

The `inverse-logical` field extension marks an inverse relationship which is not shared at the datastore level. This extension is only useful when **managed inverses** are enabled. Kodo will automatically keep the value of this field and its inverse field in synch. See [Section 5.3, "Managed Inverses" \[188\]](#).

### 6.2.1.3. dependent

---

Setting a value of `true` for the `dependent` field extension indicates that the persistent object stored in this field should be deleted when the parent object is deleted, or when the parent field is nulled.

---

#### 6.2.1.4. element-dependent

The `element-dependent` field 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.

---

#### 6.2.1.5. value-dependent

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

---

#### 6.2.1.6. key-dependent

The `key-dependent` field extension is equivalent to the `value-dependent` extension above, but is used for map keys rather than map values.

---

#### 6.2.1.7. type

Kodo JDO has three levels of support for relations:

1. Relations that hold a reference to an object of a concrete persistent class are supported by storing the primary key values of the related instance in the database.
2. Relations that hold a reference to an object of an unknown persistent class are supported by storing the stringified identity value of the related instance. This level of support does not allow queries across the relation.
3. Relations that hold an unknown object or interface. The only way to support these relations is to serialize their value to the database. This does not allow you to query the field, and is not very efficient.

Clearly, when you declare a field's type to be another persistence-capable class, Kodo JDO uses level 1 support. By default, Kodo JDO assumes that any interface-typed fields you declare will be implemented only by other persistent classes, and assigns interfaces level 2 support. The exception to this rule is the `java.io.Serializable` interface. If you declare a field to be of type `Serializable`, Kodo JDO lumps it together with `java.lang.Object` fields and other non-interface, unrecognized field types, which are all assigned level 3 support.

With the `type` field extension, you can control the level of support given to your unknown/interface-typed fields. Setting the value of this extension to `javax.jdo.spi.PersistenceCapable` -- or just `PersistenceCapable` for short -- indicates that the field value will always be some persistent object, and gives level 2 support. Setting the value of this extension to the full class name of a concrete persistent type is even better; it gives you level 1 support (just as if you had declared your field to be of that type in the first place). Setting this extension to `java.lang.Object` -- `Object` for short -- uses level 3 support. This is useful when you have an interface relation that may **not** hold other persistent objects (recall that Kodo JDO assumes interface fields will always hold persistent instances by default).

---

#### 6.2.1.8. element-type

The `element-type` field extension is equivalent to the `type` extension above, but is used for the element values of collections with generic/interface `element-type` metadata.

This extension can also be used with externalization (see [Section 7.9.23, “Externalization” \[273\]](#)) to indicate the element type if a field externalizes to a collection.

---

### 6.2.1.9. value-type

---

The `value-type` field extension is equivalent to the `type` extension above, but is used for the values of maps with generic/interface `value-type` metadata.

This extension can also be used with externalization (see [Section 7.9.23, “Externalization” \[273\]](#)) to indicate the value type if a field externalizes to a map.

### 6.2.1.10. key-type

---

The `key-type` field extension is equivalent to the `type` extension above, but is used for the keys of maps with generic/interface `key-type` metadata.

This extension can also be used with externalization (see [Section 7.9.23, “Externalization” \[273\]](#)) to indicate the key type if a field externalizes to a map.

### 6.2.1.11. lrs

---

Use this boolean field-level extension to mark fields that should use Kodo's special large result set collection or map proxies. A complete description of large result set proxies is available in [Section 5.4.3.2, “Large Result Set Proxies” \[190\]](#).

### 6.2.1.12. Example

---

```
<jdo>
  <package name="com.xyz">
    <class name="Company">
      <field name="employees">
        <collection element-type="Employee"/>
        <!-- specify the field that owns this two-sided relation -->
        <extension vendor-name="kodo" key="inverse-owner" value="employer"/>
        <!-- delete all employees when the company is deleted -->
        <extension vendor-name="kodo" key="element-dependent" value="true"/>
        <!-- very large company, thousands of employees -->
        <extension vendor-name="kodo" key="lrs" value="true"/>
      </field>
    </class>
    <class name="Employee">
      <field name="employer">
        <!-- no inverse-owner on this field, because it is the owner -->
      </field>
    </class>
  </package>
</jdo>
```

## 6.2.2. Schema Extensions

---

If you use Kodo JDO's automatic schema creation and migration through the **mapping tool**, you may want to exercise some control over nuances like column sizes, indexes, and foreign keys. Kodo JDO provides a simple set of metadata extensions you can use to optimize the schema generated for your classes. None of these extensions are used at runtime, or if you have an existing schema. They are only used when the mapping tool generates the schema for the classes the first time it runs on them.

### 6.2.2.1. jdbc-size

---

The `jdbc-size` field extension sets the size of the column used to hold the field's data. If this extension is not given, string fields default to a column size of 255, and other types use the database default. Use a value of -1 to indicate that this field is of an unlimited size (this typically translates to using a BLOB or CLOB mapping for the field).

### 6.2.2.2. jdbc-element-size

---

The `jdbc-element-size` field extension is equivalent to the `jdbc-size` extension, but applies to the data stored in each

element of a collection or array. Note that if the size is set such that Kodo JDO would use a BLOB mapping for each element, the entire collection will be collapsed into a single BLOB value instead for efficiency.

### 6.2.2.3. jdbc-value-size

---

The `jdbc-value-size` field extension is equivalent to the `jdbc-element-size` extension, but applies to map values rather than collection elements.

### 6.2.2.4. jdbc-key-size

---

The `jdbc-key-size` field extension is equivalent to the `jdbc-value-size` extension, but applies to map keys rather than values.

### 6.2.2.5. jdbc-type

---

The `jdbc-type` field extension specifies the column type for the given field. The names correlate to `java.sql.Types` constants, such as `VARCHAR` and `BIGINT`. The name type is case-insensitive. *This extension is currently only used for single column mappings.*

### 6.2.2.6. jdbc-sql-type

---

The `jdbc-sql-type` field extension specifies the concrete SQL name for the column type for the given field. This allows one to override database preferred types for a database specific datatype. Unlike `jdbc-type`, this extension relies upon the database's native type syntax. *This extension is currently only used for single column mappings.*

### 6.2.2.7. jdbc-indexed

---

The `jdbc-indexed` field extension specifies whether the column holding the data for this field should be indexed. Recognized values are `true`, `false`, and `unique`. By default, Kodo JDO does not index columns unless they hold a primary key value for a related database record (i.e. unless they are part of a foreign key, actual or logical).

### 6.2.2.8. jdbc-element-indexed

---

The `jdbc-element-indexed` field extension is equivalent to the `jdbc-indexed` extension, but applies to columns holding collection elements.

### 6.2.2.9. jdbc-value-indexed

---

The `jdbc-value-indexed` field extension is equivalent to the `jdbc-element-indexed` extension, but applies to map values rather than collection elements.

### 6.2.2.10. jdbc-key-indexed

---

The `jdbc-key-indexed` field extension is equivalent to the `jdbc-value-indexed` extension, but applies to map keys rather than values.

### 6.2.2.11. jdbc-ref-indexed

---

The `jdbc-ref-indexed` field indexing extension applies to the back-reference columns of a mapping. When the data for a mapping is in a row other than the row that holds the owning object's primary key values (a.k.a. the primary row), the back-reference columns act as a foreign key back to the primary row.

For example, map fields are typically stored in a table by themselves. The table consists of column(s) for the map key, column(s) for the map value, and back-reference column(s) that hold the owning object's primary key values, and which can be used to join back to the owning object's primary row.

Kodo JDO indexes all back reference columns by default, because they are often used in joins.

### 6.2.2.12. jdbc-version-ind-indexed

---

The `jdbc-version-ind-indexed` class extension is equivalent to the `jdbc-indexed` extension, but applies to the columns of the class' **version indicator**. Defaults to `true`.

### 6.2.2.13. jdbc-class-ind-indexed

---

The `jdbc-class-ind-indexed` class extension is equivalent to the `jdbc-indexed` extension, but applies to the columns of the class' **class indicator**. Defaults to `true`.

### 6.2.2.14. jdbc-delete-action

---

If a field holds a relation to another object, you can use the `jdbc-delete-action` field extension to control the delete action of the database foreign key that models this relation. Possible values are:

- `exception`: Do not allow the related record to be deleted until this record has been deleted.
- `exception-deferred`: Equivalent to the `exception` action, but the constraint is not evaluated until the database transaction is committed.
- `null`: Null the column(s) of this foreign key when the related record is deleted.
- `null-deferred`: Equivalent to the `null` action, but the constraint is not evaluated until the database transaction is committed.
- `default`: Set the column(s) of this foreign key to their database default values when the related record is deleted.
- `default-deferred`: Equivalent to the `default` action, but the constraint is not evaluated until the database transaction is committed.
- `cascade`: Delete this record when the related record is deleted.
- `cascade-deferred`: Equivalent to the `cascade` action, but the constraint is not evaluated until the database transaction is committed.
- `none`: Do not perform any action when the related record is deleted.

Kodo JDO defaults all relations to the `none` delete action, meaning the foreign key is only logical, and does not exist in the database. If you choose to use the `exception` action, and you choose not to use deferred foreign keys, make sure to enable Kodo JDO's statement ordering option to meet foreign key dependencies. Statement ordering is covered in **Section 4.11, “SQL Statement Ordering & Foreign Keys” [182]**.

Note that not all databases support all delete actions. If you specify an action that is not supported, the relevant foreign key will not be created. This will not have an adverse effect on Kodo JDO's runtime behavior.

Note that this extension only controls what Kodo expects the database server to do when a row is deleted: Kodo will not itself delete the foreign rows, but merely knows to expect them to be deleted. To have Kodo perform these actions (e.g., to have automatic client-side cascading delete), you should instead use the “dependent” extensions. See **Section 6.2.1.3, “dependent” [198]**.

### 6.2.2.15. jdbc-element-delete-action

---

The `jdbc-element-delete-action` field extension is equivalent to the `jdbc-delete-action` extension, but applies to collections that store related objects in each element.

---

### 6.2.2.16. jdbc-value-delete-action

The `jdbc-value-delete-action` field extension is equivalent to the `jdbc-element-delete-action` extension, but applies to maps that store related objects in each value.

### 6.2.2.17. jdbc-key-delete-action

The `jdbc-key-delete-action` field extension is equivalent to the `jdbc-value-delete-action` extension, but applies to maps that store related objects in each key.

### 6.2.2.18. jdbc-ref-delete-action

The `jdbc-ref-delete-action` field extension is equivalent to the `jdbc-delete-action` extension, but applies to the back-reference columns of a mapping. See the `jdbc-ref-indexed` description above for a discussion of back-reference columns.

If you are defining metadata for a subclass that lies in a separate table than the parent class, you can use this extension as a class-level extension control the delete action of the foreign key linking the child table records to the parent table records.

### 6.2.2.19. Example

```
<jdo>
  <package name="com.xyz">
    <class name="Company">
      <field name="commonName">
        <!-- index the company name because we search on it -->
        <extension vendor-name="kodo" key="jdbc-indexed" value="true"/>
      </field>
      <field name="description">
        <!-- this string should be unlimited length (clob) -->
        <extension vendor-name="kodo" key="jdbc-size" value="-1"/>
      </field>
      <field name="offices">
        <collection element-type="Address"/>
        <!-- remove the row in the offices xref table when the -->
        <!-- address it refers to is deleted -->
        <extension vendor-name="kodo" key="jdbc-element-delete-action" value="cascade"/>
      </field>
    </class>
  </package>
</jdo>
```

## 6.2.3. Object-Relational Mapping Extensions

Object-relational mapping is discussed in detail in [Chapter 7, \*Object-Relational Mapping\* \[209\]](#). This section only reviews extensions that give hints to the mapping tool (see [Section 7.1, “Mapping Tool” \[209\]](#)) about how to map your persistent classes and their fields, just as the schema extensions we reviewed above provide hints on how to create the schema. Most users can ignore these extensions, because they will be satisfied with Kodo JDO's defaults, or because they will explicitly specify all mapping data themselves as described in [Chapter 7, \*Object-Relational Mapping\* \[209\]](#). The extensions listed here are for intermediate users who want Kodo JDO to handle most of the mapping process, but have a few special needs. Note that the `jdbc-*` extensions below are not used at runtime, or if you have mappings already made for the relevant classes and fields. They are only used by the mapping tool when deciding how to map your objects the first time it is run on them.

### 6.2.3.1. jdbc-class-map-name

`jdbc-class-map-name`: This class extension specifies the type of **class mapping** to use for the class. The value of the extension can be either the short mapping type name, such as `flat`, `vertical`, or `horizontal`, or the full class name of the `ClassMapping` class to install. Using the full class name also allows you to specify custom class mappings that are not built in to Kodo JDO. See [Section 7.6, “Class Mapping” \[219\]](#) for details on the built-in class mappings that are available.

When mapping a persistent subclass, the value of this extension overrides the `kodo.jdbc.SubclassMapping` configuration property.

Set this extension to `none` for classes that do not need mappings because they will never be used as persistent objects, or because they will always be used in embedded persistent fields.

### 6.2.3.2. jdbc-version-ind-name

---

The `jdbc-version-ind-name` class extension specifies the type of **version indicator** to use for the class. Version indicators can only be specified for base classes. The value of the extension can be either the short mapping type name, such as `version-number` or `state-image`, or the full class name of the `VersionIndicator` class to install. Using the full class name also allows you to specify custom indicators that are not built in to Kodo JDO.

The value of this extension overrides the `kodo.jdbc.VersionIndicator` configuration property.

Specify a value of `none` to forgo a version indicator on the class. Note that when you do not use a version indicator, optimistic lock exceptions cannot be detected.

### 6.2.3.3. jdbc-class-ind-name

---

The `jdbc-class-ind-name` class extension specifies the type of class indicator (See [Section 7.8, “Class Indicator” \[232\]](#)) to use for the class. Class indicators can only be specified for base classes. The value of the extension can be either the short mapping type name, such as `in-class-name`, or the full class name of the `ClassIndicator` class to install. Using the full class name also allows you to specify custom indicators that are not built in to Kodo JDO.

The value of this extension overrides the `kodo.jdbc.ClassIndicator` configuration property.

Specify a value of `none` to forgo a class indicator on the class. Note that when you do not use a class indicator, you cannot inherit from this class with other persistent classes.

### 6.2.3.4. jdbc-field-map-name

---

The `jdbc-field-map-name` field extension specifies the type of **field mapping** to use for the field. The value of the extension can be either the short mapping type name, such as `value` or `one-one`, or the full class name of the `FieldMapping` class to install. Using the full class name also allows you to specify custom field mappings that are not built in to Kodo JDO. Use a value of `none` for all fields in classes that have a `none` class mapping.

### 6.2.3.5. jdbc-field-mappings

---

The `jdbc-field-mappings` class extension is used only when using horizontal mappings, and allows a subclass to define the concrete mappings for the fields defined in the superclass. See [Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)

### 6.2.3.6. jdbc-ordered

---

The `jdbc-ordered` field extension specifies whether special care should be taken to keep the elements of this collection field ordered when they are retrieved from the database. Typically, databases do not maintain ordering. Setting this extension to `true` will add a special ordering column to the table holding the collection elements. Kodo JDO can then use this ordering column to retrieve the collection elements in the same order they appeared in memory when they were last flushed.

This extension defaults to `true` for array and `java.util.List` fields, and `false` for all other collection types.

Only the owning side of a two-sided many-many relation can maintain collection order.

### 6.2.3.7. jdbc-container-meta

---

Container metadata is used to record non-essential information about collection and map fields. If this extension is set to `true`,

collections and maps will be able to distinguish between the empty state and the null state. If this extension is set to `false` or is unset, then it will not be possible for Kodo to differentiate between these two states. In this situation, all collections and maps in persistent objects loaded from the database will be non-null.

### 6.2.3.8. jdbc-null-ind

---

The `jdbc-null-ind` field extension is used only for **embedded one-to-one** mappings. Set the value of this extension to the name of the field in the embedded class that can be used to tell whether the embedded object is null. When loading an embedded object from the database, the system will check the column(s) for this field to see if they are null. If so, it will assume the embedded object was null when stored, and will assign null to the embedding field. Otherwise, it will instantiate an instance of the embedded class and assign the new instance to the embedding field.

The value of this extension defaults to `synthetic`, which means the system will create a special column in the embedding class' table just to hold whether or not the embedded object is null. In this configuration the system does not rely on the value of any embedded class field.

### 6.2.3.9. externalizer

---

The `externalizer` field extension names a method to transform a field value that is unsupported by JDO to one that is supported. See [Section 7.9.23, “Externalization” \[273\]](#) for details.

### 6.2.3.10. factory

---

The `factory` field extension names a method to re-create a field value from its externalized form. See [Section 7.9.23, “Externalization” \[273\]](#) for details.

### 6.2.3.11. external-values

---

The `external-values` field extension declares values for transformation of simple fields to different constant values in the datastore. See [Section 7.9.24, “External Values” \[276\]](#) for details.

### 6.2.3.12. jdbc-class-ind-value

---

The `jdbc-class-ind-value` class extension is reserved for classes that use the metadata value class indicator (see [Section 7.8.2, “Metadata Value Indicator” \[233\]](#)). This indicator requires that all classes use this extension to specify the database value that indicates a record of the owning class. See the class indicator description for details.

### 6.2.3.13. Example

---

```
<jdo>
  <package name="com.xyz">
    <class name="Company">
      <!-- we don't need to ever inherit from this type -->
      <extension vendor-name="kodo" key="jdbc-class-ind-name" value="none"/>
      <field name="offices">
        <collection element-type="Address"/>
        <!-- keep this collection ordered -->
        <extension vendor-name="kodo" key="jdbc-ordered" value="true"/>
      </field>
    </class>
    <class name="Employee" persistence-capable-superclass="Person">
      <!-- use a vertical class mapping for this subclass -->
      <extension vendor-name="kodo" key="jdbc-class-map-name" value="vertical"/>
      <field name="male">
        <!-- use a custom mapping that maps this boolean to an -->
        <!-- 'M' or 'F' value in the database -->
        <extension vendor-name="kodo" key="jdbc-field-map-name"
          value="com.xyz.BooleanToCharMapping"/>
      </field>
    </class>
  </package>
</jdo>
```

## 6.2.4. Miscellaneous Extensions

---

Kodo JDO recognizes the following miscellaneous extensions.

### 6.2.4.1. detachable

---

The `detachable` class extension indicates that this class should be detachable, as described at [Section 11.1, “Detach and Attach” \[297\]](#)

### 6.2.4.2. detached-objectid-field

---

The `detached-objectid-field` class extension defines the field that should be used to hold the object id of the persistent instance when detached. See [Section 11.1, “Detach and Attach” \[297\]](#)

### 6.2.4.3. detached-state-field

---

The `detached-state-field` class extension defines the field that should be used to hold the state of the persistent instance when detached. See [Section 11.1, “Detach and Attach” \[297\]](#)

### 6.2.4.4. fetch-group

---

The `fetch-group` field extension names a custom fetch group for the field. We discuss custom fetch groups in [Section 14.5, “Fetch Groups” \[337\]](#)

### 6.2.4.5. lock-group

---

The `lock-group` field extension names the lock group for the field, allowing fine-grained optimistic locking concurrency. We discuss lock groups in [Section 14.6, “Lock Groups” \[340\]](#)

### 6.2.4.6. lock-groups

---

The `lock-groups` class extension lists lock group names that subclasses of this class will use. This is only valid on least-derived types in an inheritance hierarchy. For details, see [Section 14.6.1, “Lock Groups and Subclasses” \[341\]](#)

### 6.2.4.7. data-cache

---

The `data-cache` class extension specifies **data cache** for this class. It can be one of the following values:

- `true`: this class should be cached in the default cache, as configured by the `kodo.DataCache` configuration parameter. This is the default value.
- `false`: this class should not be cached.
- `cache-name`: this class should be cached in the named cache called *cache-name*.

If not specified and if caching is enabled, members of the class will be stored in the default data cache.

### 6.2.4.8. data-cache-timeout

---

While the `kodo.DataCacheTimeout` configuration property sets the number of milliseconds that data in the **data cache** remains valid on a system-wide basis, the `data-cache-timeout` class extension overrides the system setting for an individual class. Use a value of -1 for no expiration. This is the default value.

### 6.2.4.9. sequence-assigned

---

The `sequence-assigned` extension indicates that this field should be assigned a value from the `SequenceGenerator` assigned to the owning class. The field must be of integer, short, long, or their wrapper types.

This is useful for creating primary key values for application identity. This value will be assigned when the owning instance is made persistent and no value had been assigned.

For further details on using this extension for primary key fields, see [Section 5.2.4.3, “Sequence-Assigned” \[188\]](#).

---

### 6.2.4.10. subclass-fetch-mode

This class-level extension determines how Kodo will select data in subclass tables. Legal values are `none`, `join`, and `parallel`. See [Section 14.2, “Eager Fetching” \[326\]](#) for a discussion of each fetch mode.

---

### 6.2.4.11. eager-fetch-mode

This field-level extension states the field's preference for how it is eagerly loaded. Note that turning on and off eager fetching for fields is a **fetch group** and runtime operation; this extension only declares what type of eager fetching to use *if* this field is eager fetched. Furthermore, this extension is only applicable to fields that represent relations or collections.

A value of `none` ensures that this field will never be eagerly fetched. It will always load in its own independent select. A value of `join` means to use a join to read this field along with the parent object data. This is the default for direct relations. For collection fields, the default is `parallel`. Parallel mode uses separate selects executed in parallel for each eager collection. [Section 14.2, “Eager Fetching” \[326\]](#) discusses each fetch mode in more detail, and describes when eager fetching takes effect.

---

### 6.2.4.12. jdbc-sequence-factory

The `jdbc-sequence-factory` class extension specifies a plugin string describing the **SequenceFactory** to use to generate unique datastore identity values for new instances of this class. If not given, the class will use the default sequence factory defined in the `kodo.jdbc.SequenceFactory` configuration property (see [Section 2.6.57, “kodo.jdbc.SequenceFactory” \[158\]](#)). For more information on sequence factories, see [Section 5.2.4.1, “Sequence Factory” \[185\]](#).

---

### 6.2.4.13. jdbc-sequence-name

When using the **ClassSequenceFactory**, the `jdbc-sequence-name` class extension contains the name of the database sequence for the class. If not given, the default sequence will be used.

---

### 6.2.4.14. jdbc-auto-increment

Placing this extension beneath a `<class>` element tells Kodo that its datastore identity primary key column is auto-incrementing. Placing it beneath a `<field>` element indicates that the field uses an auto-increment column. If you are using auto-increment columns, then you must specify this extension in the right places, because Kodo cannot reliably determine which columns are auto-incrementing through the JDBC driver alone. Please be sure to read [Section 5.2.4.2, “Auto-Increment” \[187\]](#) for additional steps you must take to use auto-incrementing primary key columns.

---

### 6.2.4.15. Example

```
<jdo>
  <package name="com.xyz">
    <class name="Company">
      <extension vendor-name="kodo" key="data-cache" value="false"/>
      <extension vendor-name="kodo" key="jdbc-sequence-name" value="COMP"/>
    </class>
    <class name="Person">
      <extension vendor-name="kodo" key="jdbc-auto-increment" value="true"/>
    </class>
    <class name="Form" objectid-class="FormId">
      <field name="id" primary-key="true">
        <extension vendor-name="kodo" key="jdbc-auto-increment" value="true"/>
      </field>
    </class>
  </package>
</jdo>
```



---

# Chapter 7. Object-Relational Mapping

Object-relational mapping is the process of mapping software objects to relational database tables. Kodo JDO has a full-featured mapping system including built-in support for most object-oriented patterns and schema designs. You are not limited to what Kodo JDO bundles by default, however. Kodo JDO's mapping system is designed with flexibility and extensibility in mind, so you can easily create custom mappings to meet your exact needs. This chapter reviews the mapping utilities Kodo JDO provides, the built-in mappings it supports, and how to create your own mappings should the need arise.

## 7.1. Mapping Tool

---

Kodo JDO allows you to control the mapping process yourself, but it also provides tools to automate mapping. We already saw one example of Kodo JDO's object-relational mapping tools when we discussed the **reverse mapping tool**. In this section, we discuss another mapping utility, simply called the *mapping tool*. While the reverse mapping tool creates classes and mapping data from an existing schema, the mapping tool creates the schema and mapping data from existing classes. The mapping tool can also be used to validate mapping data that you've written yourself, or to import and export mapping data to and from the current **mapping factory**.

We describe common mapping tool use cases in the next section. You can invoke the mapping tool through the `mappingtool` shell/batch script included in the Kodo JDO distribution, or through its Java class, `kodo.jdbc.meta.MappingTool`.

### *Example 7.1. Using the Mapping Tool*

```
mappingtool -a refresh *.jdo
```

In addition to the universal flags of the **configuration framework**, the mapping tool accepts the following command line arguments:

- `-file/-f <stdout | output file>`: Use this option to write the planned mappings to an XML document rather than recording them as the mappings for the given classes. This option also specifies the file to dump to if using the `export` tool action.
- `-schemaAction/-sa <add | refresh | build | retain | none>`: The action to take on the schema. These options correspond to the same-named actions on the schema tool described in [Section 8.2, “Schema Tool” \[280\]](#). Unless you are running the mapping tool on all of your persistent types at once, we strongly recommend you use the default `add` action or the `build` action. Otherwise you may end up inadvertently dropping schema components that are used by classes you are not currently running the tool over.
- `-schemaFile/-sf <stdout | output file>`: Use this option to write the planned schema to an XML document rather than modify the data store. The document can then be manipulated and committed to the database with the **schema tool**.
- `-sqlFile/-sql <stdout | output file>`: Use this option to write the planned schema modifications to a SQL script rather than modify the data store. Combine this with a `schemaAction` of `build` to generate a script that recreates the schema for the current mappings, even if the schema already exists.
- `-dropTables/-dt <true/t | false/f>`: Corresponds to the same-named option on the schema tool.
- `-ignoreErrors/-i <true/t | false/f>`: Corresponds to the same-named option on the schema tool.
- `-schemas/-s <schema and table names>`: Corresponds to the same-named option on the schema tool. This op-

tion is ignored if `readSchema` is not set to `true`.

- `-readSchema/-rs <true/t | false/f>`: Set this option to `true` to read the entire existing schema when the tool runs. Reading the existing schema ensures that Kodo does not generate any mappings that use table, index, primary key, or foreign key names that conflict with existing names. Depending on the JDBC driver, though, it can be a very slow process for large schemas.
- `-primaryKeys/-pk <true/t | false/f>`: Whether to read and manipulate primary key information of existing tables. Defaults to `false` unless the `readSchema` flag is set to `true`.
- `-foreignKeys/-fk <true/t | false/f>`: Whether to read and manipulate foreign key information of existing tables. Defaults to `false` unless the `readSchema` flag is set to `true`. This means that if you add a **`jdbc-delete-action`** extension to a field of a class that has already been mapped once, you must explicitly set this flag to `true` to have Kodo create the new foreign key on the existing table.
- `-indexes/-ix <true/t | false/f>`: Whether to read and manipulate index information of existing tables. Defaults to `false` unless the `readSchema` flag is set to `true`. This means that if you add a **`jdbc-indexed`** extension to a field of a class that has already been mapped once, you must explicitly set this flag to `true` to have Kodo create the new index on the existing table.

The mapping tool requires an `-action/-a` argument specifying the action to take on each class. The available actions are:

- **`refresh`**: Bring the mapping information up-to-date with the class definitions. Classes or fields whose mappings no longer match the class definition or schema will be re-mapped to new columns/tables.
- **`drop`**: Remove the mapping information for the given classes.
- **`validate`**: Validate that the mappings for the given classes are valid and that they match the schema. No mappings or tables will be changed; an exception will be thrown if any mappings are invalid.
- **`buildSchema`**: Create the schema based on the existing mappings for the given classes.
- **`revert`**: Revert the mappings for the given classes to their previously saved state. Some mapping factories may not be able to revert mapping data.
- **`import`**: Import mapping information from the given XML document and add it to the stored system mappings. The XML format used for mapping data is discussed in [Section 7.4, “Mapping File XML Format” \[215\]](#)
- **`export`**: Export the mapping data for the given classes to an XML file. The XML format used for mapping data is discussed in [Section 7.4, “Mapping File XML Format” \[215\]](#)

Each additional argument to the tool should be either the full name of a persistent class, the `.java` file of a persistent class, the `.class` file of a persistent class, or a `.jdo` metadata file listing one or more persistent classes to act on. If the `import` action is used, however, then any additional arguments will be interpreted as **XML mapping data** files.

The mapping data generated by the mapping tool is stored in the system **mapping factory**. As you will see later in this chapter, you have several mapping factories to choose from. Thus, mapping data might end up stored in the database, in special mapping files, in JDO metadata vendor extensions, or in another format of your choosing.

---

### 7.1.1. Using the Mapping Tool

There are three primary approaches to object-relational mapping: *object-to-schema*, *schema-to-object*, and *meet-in-the-middle*. The mapping tool has actions to facilitate each approach.

In the *object-to-schema* approach to mapping, you concentrate your efforts on your object model, and the mapping tool's `refresh` action keeps your mappings and schema up-to-date. The `refresh` action examines both the existing database schema and

any existing mapping information. Classes and fields that are not mapped, or whose mapping information no longer matches the object model or the schema, are automatically given new mappings. The tool also updates the schema as necessary to support these new mappings. The example below shows how to invoke the refresh action on the mapping tool to create or update the mapping information and database schema for the persistent classes listed in package .jdo.

### ***Example 7.2. Refreshing Mappings and the Relational Schema***

```
mappingtool -a refresh package.jdo
```

You can safely run the `refresh` action on classes that have already been mapped, because the tool only generates new mappings when the old ones have become incompatible with the class or the schema. If the tool does have to replace a bad mapping, it does not modify other still-valid mappings. For example, if you change the type of a field from `int` to `String`, the mapping tool will detect the incompatibility with the old numeric column, add a new string-compatible column to the class' database table, and change the field's mapping data to point to the new column. All other fields will retain their original mappings.

To drop mapping data, use the `drop` action. This action does not affect the schema. Dropping mapping data for unused classes is not strictly necessary, but it might slightly increase performance under some **mapping factories**.

### ***Example 7.3. Dropping Mappings***

```
mappingtool -a drop package.jdo
```

The second approach to object-relational mapping is the *schema-to-object* approach. We have already seen how to use the **reverse mapping tool** to generate persistent classes and mapping information from an existing schema. Once you complete the reverse mapping, you may want to tweak the output of the reverse mapping tool. At this point you have both an existing schema and existing mapping information (from the reverse mapping tool), and you are modifying both by hand. Thus, you are really using the final, meet-in-the-middle approach to mapping.

In the *meet-in-the-middle* mapping approach, you control both the relational model and the object model. It is up to you to define the mappings between these models, possibly with the aid of Kodo JDO's GUI tools. In this scenario, you will find the mapping tool's `validate` action useful. The `validate` action verifies that the mapping information for a class matches the class definition and the existing schema.

### ***Example 7.4. Validating Mappings***

```
mappingtool -a validate package.jdo
```

The `buildSchema` tool action is also useful for meet-in-the-middle mapping. Unlike the `validate` action, which throws an exception if the mapping data does not match the existing schema, or the `refresh` action, which replaces inconsistent mappings, the `buildSchema` action assumes your mapping data is correct, and modifies the schema to match your mappings. This lets you modify your mapping data manually, but saves you the hassle of using your database's tools to bring the schema up-

to-date.

### ***Example 7.5. Updating the Schema Based on Mapping Data***

```
mappingtool -a buildSchema package.jdo
```

The `buildSchema` action is also useful if you would like Kodo to do most of your mappings, but you want to edit a few of the mappings or table/column names Kodo generates:

### ***Example 7.6. Modifying Default Mappings***

First, run the mapping tool with a `none` schema action to generate default mappings without changing the database:

```
mappingtool -a refresh -sa none package.jdo
```

Next, modify the default mappings to fit your needs. You only have to do this once; Kodo will continue to use the modified mappings as long as they remain valid. Finally, build the schema based on your mappings:

```
mappingtool -a buildSchema package.jdo
```

Finally, some **mapping factories** allow you to revert mapping data if they have saved a copy.

### ***Example 7.7. Reverting Mapping Data***

```
mappingtool -a revert package.jdo
```

---

## **7.1.2. Generating DDL SQL**

Sometimes you do not want Kodo to make the changes to your database, but rather you want to have a script of the SQL that Kodo would use. The examples below show how to use the mapping tool to generate DDL SQL scripts.

### ***Example 7.8. Refresh Mappings and Create DDL***

This example refreshes the mappings for the classes in `package.jdo` and writes all the SQL necessary to create the tables used by these mappings to `create.sql`.

```
mappingtool -a refresh -sa build -sql create.sql package.jdo
```

### *Example 7.9. Refresh Mappings and Create DDL to Update Database*

This example refreshes the mappings for the classes in `package.jdo`. It writes the SQL to add tables or columns missing from the current schema to the `update.sql` file.

```
mappingtool -a refresh -sql update.sql package.jdo
```

### *Example 7.10. Create DDL for Current Mappings*

This example uses your existing mappings to determine the needed schema, then writes the SQL to create that schema to `create.sql`.

```
mappingtool -a buildSchema -sa build -sql create.sql package.jdo
```

### *Example 7.11. Create DDL to Update Database for Current Mappings*

This example uses your existing mappings to determine the needed schema. It then writes the SQL to add any missing tables and columns to the current schema to `update.sql`.

```
mappingtool -a buildSchema -sql update.sql package.jdo
```

## 7.2. Automatic Runtime Mapping

---

You can configure Kodo to automatically create O/R mapping information and the database schema at runtime through the **`kodo.jdbc.SynchronizeMappings`** configuration property. Using this property saves you the trouble of running the mapping tool manually, and is meant for use during rapid test/debug cycles.

In order to enable automatic runtime mapping, you must first set the **`kodo.PersistentClasses`** property to include all the persistent classes. Kodo will run the mapping tool on these classes when your application obtains its first persistence manager.

The `kodo.jdbc.SynchronizeSchema` configuration property accepts the following values:

- `false`: The default. No runtime mapping synchronization is performed.
- `refresh`: Invoke the mapping tool with the refresh action, which throws out all existing O/R mapping data that doesn't match the current schema, creates new O/R mappings for unmapped classes and fields, and modifies the schema to match the new mappings.
- `buildSchema`: Invoke the mapping tool with the buildSchema action. This action requires that all your classes and fields already have mapping information, and synchronizes the database schema to match that information.

## 7.3. Mapping Factory

---

An important decision in the object-relational mapping process is how and where to store the data necessary to map your persistent classes to the database schema. If you rely on the **mapping tool** to do all your mapping for you, you might want to keep mapping data out of the way in a database table. On the other hand, if you want easy access to your mapping information, or if you do not want to store any additional metadata in your database, you might want to store it as vendor extensions in your JDO metadata. Or perhaps JDO's metadata extension mechanism is too verbose for your tastes, and you'd like to use separate, more concise mapping files to express your mappings.

Kodo JDO uses the `kodo.jdbc.meta.MappingFactory` interface to abstract the storage and retrieval of mapping information. Kodo JDO includes built-in mapping factories for all of the options mentioned above, and you can create your own factory if you have custom needs. You control which mapping factory Kodo JDO uses with the `kodo.jdbc.MappingFactory` configuration property.

The bundled mapping factories are:

- `file`: This is the default mapping factory. It is an alias for the `kodo.jdbc.meta.FileMappingFactory`. As its name implies, the `FileMappingFactory` stores mapping data in the file system. The data is stored in an **XML format** that closely resembles the JDO metadata format, and the placement of mapping files also follows the rules for the **placement of JDO metadata files**, the only difference being that mapping files use the `.mapping` extension rather than the `.jdo` extension.

The main advantages of this mapping factory are that it allows easy access to mapping data and that it doesn't create any special database tables. Additionally, its concise XML format is easier to manipulate than JDO metadata extensions, which are another option for mapping information storage (see below).

The file mapping factory accepts the following properties:

- `SingleFile`: Set this property to true to have all mapping information stored in a single file. By default, the factory creates a mapping file for each JDO metadata file.
- `FileName`: If you are using single file mode, then this property specifies the resource name of the XML mapping file. By default, the factory looks for a resource called `package.mapping`, located in any top-level directory of the CLASSPATH or in the top level of any jar.
- `metadata`: This is an alias for the `kodo.jdbc.meta.MetadataMappingFactory`: The `MetadataMappingFactory` stores mapping data in your `.jdo` files using JDO metadata's built-in extension mechanism. This allows you easy access to the mapping information, and it consolidates all of your JDO metadata information in one place.

The only major disadvantage to using JDO metadata extensions is that they are rather verbose. For that reason some users may prefer the `FileMappingFactory` over this one.

- `db`: This option is appropriate if you want Kodo JDO to do all the mapping work for you through the mapping tool (see **Section 7.1, "Mapping Tool" [209]**). It is an alias for the `kodo.jdbc.meta.DBMappingFactory`. The `DBMappingFactory` stores its mapping data in a special database table it creates the first time the factory is used. Storing the mapping data in the database means you never need to see it or deal with it. It also means, however, that you can't access your mapping

data easily if you want to manipulate it by hand. Recall that there are still **metadata extensions** you can use to control general mapping options like which columns are indexed or even what type of mapping to create on a given field. But if you plan on any detailed hand-mapping, you should use one of the other factories presented above. This factory accepts the following properties:

- `SingleRow`: Set this property to true to have all mapping information stored in a single table row. By default, the factory creates a row for the mapping data of each class.
- `TableName`: The name of the table to create to store mapping data. Defaults to `JDO_MAPPING`.

Note that using a mapping factory other than the `MetaDataMappingFactory` does not obviate the need for JDO metadata extensions. Extensions such as **dependent**, **inverse-owner**, and, if you use Kodo to create your schema, **jdbc-size** and **jdbc-indexed** still reside in your JDO metadata. While some of these extensions may affect object/relational mapping behavior, they do not contain object/relational mapping data per se. Mapping factories only hold information directly related to object/relational mapping: which columns a field occupies, and how those columns are linked to other schema components.

### 7.3.1. Importing and Exporting Mapping Data

---

The **mapping tool** has the ability to import object-relational mapping data into the mapping factory, and to export mapping data from the mapping factory. We discuss the XML format used for imports and exports [here](#).

Importing and exporting mapping data is useful for a couple of reasons. First, you may want to use a mapping factory that stores mapping data in an out-of-the-way location like the database, but you still want the ability to manipulate this information occasionally by hand. You can do so by exporting the data to XML, modifying it, and then re-importing it.

#### *Example 7.12. Modifying Difficult-to-Access Mapping Data*

```
mappingtool -a export -f mappings.xml package.jdo
... modify mappings.xml file as necessary ...
mappingtool -a import mappings.xml
```

Second, you can use the export/import facilities to switch mapping factories at any time.

#### *Example 7.13. Switching Mapping Factories*

```
mappingtool -a export -f mappings.xml *.jdo
... switch the kodo.jdbc.MappingFactory configuration ...
... property to list your new mapping factory choice ...
mappingtool -a import mappings.xml
```

## 7.4. Mapping File XML Format

---

Several **mapping factories** store their mapping data in XML. The **mapping tool** also serializes mapping data to and from XML during its `import` and `export` operations. This section covers the common XML format used by all of these components.

Below we present the Document Type Definition (DTD) for the XML mapping format. Note that this DTD is not valid, because in many places we declare an `ATTLIST` with a value of `ANY`, which is not legal DTD syntax. We do this to indicate that the corresponding element can have any additional attributes. Which attributes are used depends on the type of the mapping. Thus, XML mapping documents are not validated by Kodo JDO.

```
<!ELEMENT mapping (package)+>
<!ELEMENT package (class)+>
<!ATTLIST package name CDATA #REQUIRED>
<!ELEMENT class (jdbc-class-map, (jdbc-version-ind)?, (jdbc-class-ind)?,
  (field)*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ELEMENT jdbc-class-map EMPTY>
<!ATTLIST jdbc-class-map type CDATA #REQUIRED>
<!ATTLIST jdbc-class-map ANY>
<!ELEMENT jdbc-version-ind EMPTY>
<!ATTLIST jdbc-version-ind type CDATA #REQUIRED>
<!ATTLIST jdbc-version-ind ANY>
<!ELEMENT jdbc-class-ind EMPTY>
<!ATTLIST jdbc-class-ind type CDATA #REQUIRED>
<!ATTLIST jdbc-class-ind ANY>
<!ELEMENT field (jdbc-field-map)*>
<!ATTLIST field name CDATA #REQUIRED>
<!ELEMENT jdbc-field-map (field)*>
<!ATTLIST jdbc-field-map type CDATA #REQUIRED>
<!ATTLIST jdbc-field-map ANY>
```

As you can see, the format of mapping files is closely aligned with the format of **JDO metadata** files. The basic structure is the same:

### Example 7.14. Basic Structure of Mapping Documents

```
<?xml version="1.0"?>
<mapping>
  <package name="org.mag">
    <class name="Magazine">
      ... class-level data ...
      <field name="isbn">
        ... field-level data ...
      </field>
      <field name="title">
        ... field-level data ...
      </field>
      <field name="articles">
        ... field-level data ...
      </field>
    </class>
    ... other classes ...
  </package>
  ... other packages ...
</mapping>
```

Other than package, class, and field names, however, mapping documents do not repeat any information that is already found in the JDO metadata. You do not specify things like the identity type of classes, or the element type of collection fields.

Mapping documents, do, however, contain extra information not present in JDO metadata. At the class level, they contain a required `jdbc-class-map` element describing how the class is mapped to the database. The `class` element can also have optional `jdbc-version-ind` and `jdbc-class-ind` child elements describing the *version indicator* and *class indicator* mappings for the class, respectively. All of these elements have a required `type` attribute specifying the short type name of the map-

ping. You can also supply the full class name of a custom mapping type in this attribute. Class mappings, version indicators, and class indicators are covered in later sections of this chapter.

Each `field` element of a mapping document represents a member field, and all managed fields in each class must be listed. Each `field` contains a single `jdbc-field-map` element. The attributes of this element describe how the field maps to the database. Like the `jdbc-class-map`, `jdbc-version-ind`, and `jdbc-class-ind` elements, the `jdbc-field-map` element requires a `type` attribute. This attribute specifies the short type name of the mapping, or the full class name of a custom mapping. Field mappings are also discussed in depth later in this chapter.

### Note

In mappings for embedded objects, the `jdbc-field-map` elements have `field` child elements for the persistent fields of the embedded type. See the description of the **embedded one-to-one mapping**.

### Example 7.15. Complete Mapping Document

```
<?xml version="1.0"?>
<mapping>
  <package name="org.mag">
    <class name="Magazine">
      <jdbc-class-map type="base" table="MAGAZINE" pk-column="JDOID"/>
      <jdbc-version-ind type="version-number" column="JDOVERSION"/>
      <jdbc-class-ind type="in-class-name" column="JDOCLASS"/>
      <field name="isbn">
        <jdbc-field-map type="value" column="ISBN"/>
      </field>
      <field name="title">
        <jdbc-field-map type="value" column="TITLE"/>
      </field>
      <field name="articles">
        <jdbc-field-map type="one-many" table="ARTICLE"
          ref-column.JDOID="MAGAZINE_ID"/>
      </field>
    </class>
  </package>
</mapping>
```

## 7.5. Mapping Notes

---

Kodo JDO divides mappings into four functional categories: *class mappings*, *version indicators*, *class indicators*, and *field mappings*. The remainder of this chapter explores each category in turn. Examples are given for all of the mapping types Kodo JDO supports. These examples present the following information:

- A sample Java class definition supporting the relevant mapping.
- The hypothetical schema being used. The schema is given in Kodo JDO's **XML format** for schema information.
- JDO metadata for the mapping's class or field.
- The mapping information in the **XML format** used by the **file mapping factory** and other mapping tools. Remember that the **mapping tool** generally writes this mapping information for you. The examples include this information so that you can create and modify mappings by hand if you so desire.
- The same mapping information, presented as JDO metadata extensions used by the **metadata mapping factory**. This is for users who have chosen to use the metadata mapping factory, and want to understand mappings so they can create and modify them by hand.

## 7.5.1. Join Attributes

Many mappings have attributes that join one column to another. For example, if persistent class `Person` has a field of persistent type `Address`, you might model the relationship using a **one-to-one** mapping. In this mapping, the `PERSON` table includes columns to hold the primary key values of each related address. To retrieve a person's address, Kodo joins these `PERSON` table columns to the primary key columns of the `ADDRESS` table, as depicted below.

```
<table name="PERSON">
  <pk column="ID"/>
  <column name="ID" type="bigint" not-null="true"/>
  <column name="ADDRESS_PK1" type="integer"/>
  <column name="ADDRESS_PK2" type="varchar" size="255"/>
  <fk to-table="ADDRESS">
    <join column="ADDRESS_PK1" to-column="PK1"/>
    <join column="ADDRESS_PK2" to-column="PK2"/>
  </fk>
  ... other columns ...
</table>
<table name="ADDRESS">
  <pk>
    <on column="PK1"/>
    <on column="PK2"/>
  </pk>
  <column name="PK1" type="integer" not-null="true"/>
  <column name="PK2" type="varchar" size="255" not-null="true"/>
  ... other columns ...
</table>

SELECT ADDRESS.PK1, ADDRESS.PK2, ...
FROM PERSON INNER JOIN ADDRESS
  ON PERSON.ADDRESS_PK1 = ADDRESS.PK1
 AND PERSON.ADDRESS_PK2 = ADDRESS.PK2
WHERE PERSON.ID = ?
```

Mappings that use joins must record these joins in their XML representations. To accomplish this, they use XML attributes of the form `<attribute name>.<pk-column>=<local column>`. For relationship columns, the attribute name is typically just `column`. Therefore, the attributes representing the joins in our example would be:

- `column.PK1="ADDRESS_PK1"`
- `column.PK2="ADDRESS_PK2"`

When reading these attributes to yourself, replacing the `'.'` with the word `"for"` and the `'='` with the word `"is"` may help you understand their meaning. Thus, the attributes above become:

- The `column` *for* `PK1` *is* `ADDRESS_PK1`.
- The `column` *for* `PK2` *is* `ADDRESS_PK2`.

Below we present the complete mapping XML for our one-to-one field. All mappings are detailed later in this chapter.

```
<jdbc-field-map type="one-one" column.PK1="ADDRESS_PK1"
  column.PK2="ADDRESS_PK2"/>
```

## 7.5.2. Non-Standard Joins

The example in the previous section uses a "standard" join, in that there is one `PERSON` table column for each primary key column in the `ADDRESS` table. Kodo does, however, support other join patterns, including partial primary key joins, non-primary

key joins, and joins using constant values.

In a partial primary key join, the local table only has columns for a subset of the primary key columns in the table it joins to. So long as this subset of columns correctly identifies the proper row(s) in the referenced table, Kodo will function properly. There is no special syntax for expressing a partial primary key join in Kodo's XML mapping format -- just do not include XML attributes for primary key columns that are not used in the join.

In a non-primary key join, at least one of the columns being joined to is not a primary key. Once again, Kodo supports this join type with the same syntax as a primary key join. There is one restriction, however: each non-primary key column you are joining to must be controlled by a **field mapping** that implements the `kodo.jdbc.meta.JoinableMapping` interface. The built-in **value** mapping implements this interface, meaning that any column mapped to a primitive/primitive wrapper/Date/string can be joined to. Kodo will also respect any custom mappings that implement this interface.

Not all joins consist of only links between columns. In some cases you might have a schema in which one of the join criterions is that a column in the referenced table must have some constant value. Lets modify our previous person/address example to use a constant value join. The sample below depicts modified versions of the PERSON and ADDRESS tables and the corresponding SQL to retrieve a person's address. Notice the use of the ' P ' constant.

```
<table name="PERSON">
  <pk column="ID"/>
  <column name="ID" type="bigint" not-null="true"/>
  <column name="ADDRESS_ID" type="integer"/>
  <fk to-table="ADDRESS">
    <join column="ADDRESS_ID" to-column="ADDRID"/>
    <join value="' P ' " to-column="TYPE"/>
  </fk>
  ... other columns ...
</table>
<table name="ADDRESS">
  <pk>
    <on column="ADDRID"/>
    <on column="TYPE"/>
  </pk>
  <!-- the pk is made up of an id integer and an address type, which -->
  <!-- can be set to either 'P' for a person's address, or 'C' for -->
  <!-- a company's address -->
  <column name="ADDRID" type="integer" not-null="true"/>
  <column name="TYPE" type="char" size="1" not-null="true"/>
  ... other columns ...
</table>

SELECT ADDRESS.ADDRID, ADDRESS.TYPE, ...
FROM PERSON INNER JOIN ADDRESS
  ON PERSON.ADDRESS_ID = ADDRESS.ADDRID
 AND ADDRESS.TYPE = ' P '
WHERE PERSON.ID = ?
```

To express these conditions as XML join attributes, simply write the appropriate constant value as the attribute value in the now-familiar syntax:

- `column.ADDRID="ADDRESS_ID"`
- `column.TYPE=" ' P ' "`

Constant join values can be either strings or numbers. If the value is a string, be sure to place single quotes around it as we did above.

## 7.6. Class Mapping

A *class mapping* describes how a class maps to the database. It typically controls the primary table for the class and how the class is linked to its superclass data, if any. For classes using datastore identity, the class mapping also manages the primary key column for the class.

In Kodo JDO, class mappings extend the base `kodo.jdbc.meta.ClassMapping` class. The concrete class mappings Kodo

JDO provides are described in the sections below. By default, the **mapping tool** uses the **base** class mapping for all persistent classes without a persistence-capable superclass, and the **flat** class mapping for all persistent subclasses. You can change the default subclass mapping type with the `kodo.jdbc.SubclassMapping` configuration property. You can also instruct the mapping tool to use a specific mapping for an individual class with the `jdbc-class-map-name` JDO metadata extension.

## 7.6.1. Base Mapping

---

The base class mapping is reserved for persistent classes that do not extend from any other persistent class. The base class mapping has the following attributes:

- `type`: base
- `table`: The name of the table in which the primary key data is stored for each record of this class. This property is required.
- `pk-column`: The name of the primary key column for classes that use datastore identity. This property is not used for classes with application identity. The named column must be of some numeric type, as Kodo JDO uses Java `long` values for datastore identities.

### *Example 7.16. Using a Base Mapping*

```
Java class:
public class Magazine
{
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
  ... field metadata ...
</class>

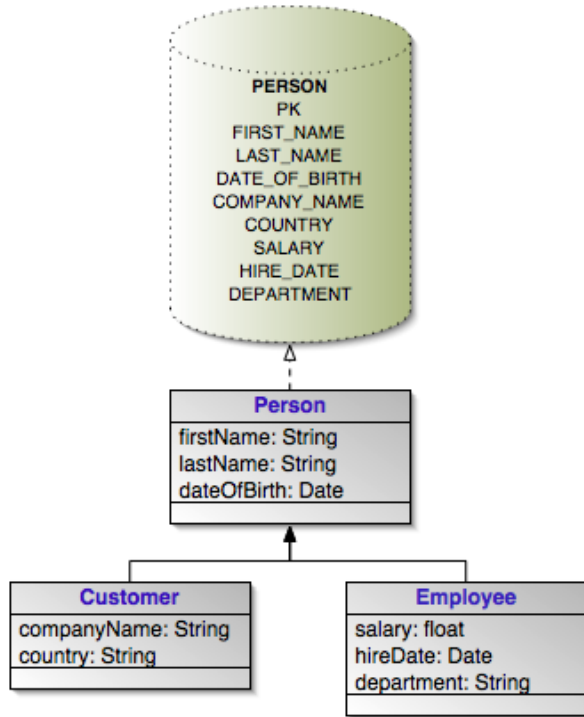
Mapping information using the mapping XML format:
<class name="Magazine">
  <jdbc-class-map type="base" table="MAGAZINE" pk-column="JDOID"/>
  ... indicator mappings ...
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
  <extension vendor-name="kodo" key="jdbc-class-map" value="base">
    <extension vendor-name="kodo" key="table" value="MAGAZINE"/>
    <extension vendor-name="kodo" key="pk-column" value="JDOID"/>
  </extension>
  ... indicator extensions ...
  ... field metadata ...
</class>
```

## 7.6.2. Flat Inheritance Mapping

---

The *flat* inheritance mapping (sometimes referred to as *single-table* or *filtered* inheritance) is a mapping for persistent subclasses that stores their fields in the same table as the parent class.



The flat class mapping has the following attributes:

- type: flat

### Example 7.17. Using a Flat Mapping

```

Java class:
public class Tabloid
    extends Magazine
{
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns ...
    ... columns for magazine fields ...
    ... columns for tabloid fields ...
</table>

JDO metadata:
<class name="Tabloid" persistence-capable-superclass="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Tabloid">
    <jdbc-class-map type="flat"/>
    ... indicator mappings ...
    ... field mappings ...
</class>
  
```

Mapping information using JDO metadata extensions:

```
<class name="Tabloid" persistence-capable-superclass="Magazine">
  <extension vendor-name="kodo" key="jdbc-class-map" value="flat"/>
  ... indicator extensions ...
  ... field metadata ...
</class>
```

### 7.6.2.1. Advantages of using Flat Inheritance Mapping

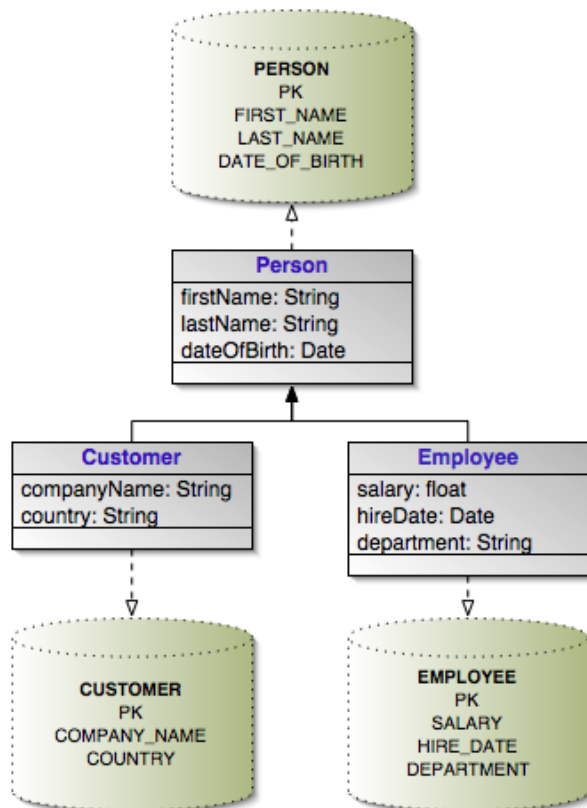
Flat inheritance mapping is the fastest of all inheritance models, since it never requires a join to retrieve a single persistent instance from the database. Similarly, persisting or updating a single persistent instance requires only a single INSERT or UPDATE statement (unlike vertical inheritance mapping).

### 7.6.2.2. Disadvantages of using Flat Inheritance Mapping

The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a deep inheritance hierarchy will result in tables with many columns.

## 7.6.3. Vertical Inheritance Mapping

Subclasses whose derived fields are in a different table than their superclass fields use a vertical class mapping.



The vertical class mapping has the following attributes:

- `type: vertical`
- `table`: The name of the table in which the derived fields of the class are stored. This property is required.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this class' table to the table of the parent class. Each `ref-column` attribute joins a column in this class' table to the corresponding column in the parent class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.

### *Example 7.18. Using a Vertical Mapping*

```
Java class:
public class Tabloid
{
    extends Magazine
    {
        ... class content ...
    }
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="TABLOID">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  ... columns for tabloid fields ...
</table>

JDO metadata:
<class name="Tabloid" persistence-capable-superclass="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Tabloid">
  <jdbc-class-map type="vertical" table="TABLOID" ref-column.JDOID="MAG_ID"/>
  ... indicator mappings ...
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Tabloid" persistence-capable-superclass="Magazine">
  <extension vendor-name="kodo" key="jdbc-class-map" value="vertical">
    <extension vendor-name="kodo" key="table" value="TABLOID"/>
    <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
  </extension>
  ... indicator extensions ...
  ... field metadata ...
</class>
```

---

## 7.6.3.1. Advantages of using Vertical Inheritance Mapping

Vertical inheritance mapping results in the most "normalized" database schema, since spurious and redundant data will not exist

in any of the tables. As more subclasses are added to the data model over time, the only schema modification that needs to be made is the addition of a corresponding table in the database (rather than having to change the structure of existing tables).

### 7.6.3.2. Disadvantages of using Vertical Inheritance Mapping

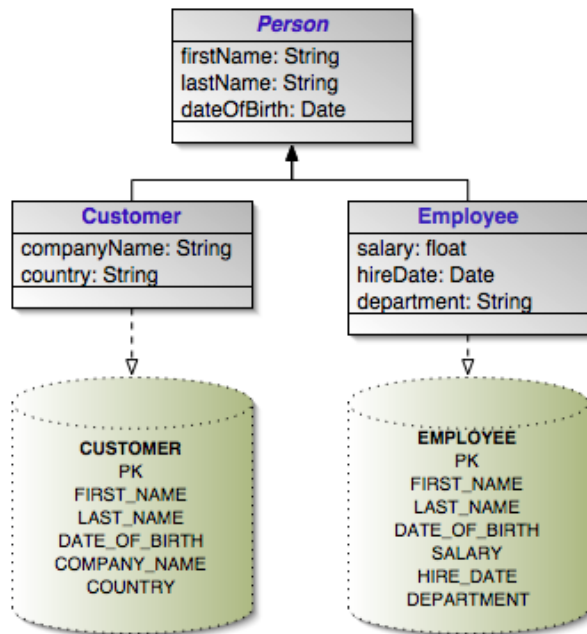
Vertical mappings can be the slowest of all the inheritance models, since retrieving any subclass will require one or more database JOINS, and storing subclasses will require multiple INSERT or UPDATE statements. This is only the case when persistence operations are performed on subclasses; if most operations are performed on the topmost persistent superclass, then a vertical mapping will be as fast as a flat mapping.

### 7.6.3.3. Vertical Select Modes

When executing a select against a hierarchy that uses vertical inheritance, you must consider how to load subclass data. [Section 14.2, “Eager Fetching”](#) [326] describes your options.

## 7.6.4. Horizontal Inheritance Mapping

The *horizontal* inheritance mapping (sometimes referred to as *distributed* or *one-table-per-leaf* inheritance) is a mapping that maps each concrete class in an inheritance hierarchy to a separate table. A horizontally mapped class does not itself have a table; all the fields in the class will be mapped by the subclasses. Therefore, a horizontally mapped class cannot be directly persisted, only subclasses of the class can be stored in the database. For this reason, it is recommended (but not required) that horizontally mapped classes be declared abstract.



When defining mappings for horizontal subclasses, the field mappings of the superclass' fields are defined in the mapping definition for the subclass. When the mappings are defined in the `.mapping` format, field mappings for superclasses will be named `"package.name.to.SuperClass.fieldName"`, or, if the superclass is in the same package as the subclass that defines its field, this can be shortened to: `"SuperClass.fieldName"`.

When defining the mappings for a horizontal superclass in a `.jdo` metadata file, the field mapping definitions must be contained in a special `"jdbc-field-mappings"` extension element, since the `.jdo` format disallows `<field>` tags to be specified for field names that are not declared by the owning class.

As well as having support for a most-derived horizontal mapping, Kodo can also have intermediate classes in any inheritance hierarchy (either vertical or flat) be declared as horizontal. This is convenient for abstract persistent classes for which no table is to be defined.

**Example 7.19. Using a Horizontal Mapping**

```
Java class:
public class Magazine
{
    private String title;
    ... class content ...
}

JDO metadata:
<class name="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    <jdbc-class-map type="horizontal"/>
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
    <extension vendor-name="kodo" key="jdbc-class-map" value="horizontal"/>
    ... field metadata ...
</class>
```

```
Java class:
public class Tabloid
    extends Magazine
{
    private String tabloidType;
    ... class content ...
}

Schema:
<table name="TABLOID">
    <pk column="JDOID"/>
    <column name="JDOID" type="bigint"/>
    <column name="TABLOID_TYPE" type="varchar" size="255"/>
    <column name="TITLE" type="varchar" size="255"/>
    ... columns for other tabloid fields ...
</table>

JDO metadata:
<class name="Tabloid" persistence-capable-superclass="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Tabloid">
    <jdbc-class-map type="base" table="TABLOID"/>
    ... indicator mappings ...
    <field name="tabloidType">
        <jdbc-field-map type="value" column="TABLOID_TYPE"/>
    </field>
    <field name="Magazine.title">
        <jdbc-field-map type="value" column="TITLE"/>
    </field>
    ... other field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Tabloid" persistence-capable-superclass="Magazine">
    <extension vendor-name="kodo" key="jdbc-class-map" value="base">
```

```

    <extension vendor-name="kodo" key="table" value="TABLOID"/>
  </extension>
  ... indicator extensions ...
  <field name="tabloidType">
    <extension vendor-name="kodo" key="jdbc-field-map" value="value">
      <extension vendor-name="kodo" key="column" value="TABLOID_TYPE"/>
    </extension>
  </field>
  <extension vendor-name="kodo" key="jdbc-field-mappings">
    <extension vendor-name="kodo" key="Magazine.title">
      <extension vendor-name="kodo" key="jdbc-field-map" value="value">
        <extension vendor-name="kodo" key="column" value="TITLE"/>
      </extension>
    </extension>
  </extension>
</class>

```

### *Example 7.20. Using a Horizontal Mapping with Application Identity hierarchy*

```

Java class:
public class Magazine
{
    private int id;
    private String title;
    ... class content ...

    // application identity class for Manager; represented as
    // a static inner class for the Magazine class
    public static class MagazineID
    {
        public int id;

        public MagazineID ()
        {
        }

        public MagazineID (String idString)
        {
            id = new Integer (idString).intValue ();
        }

        public int hashCode ()
        {
            return id;
        }

        public boolean equals (Object other)
        {
            if (other == null || other.getClass () != getClass ())
                return false;

            return ((MagazineID)other).id == id;
        }
    }
}

```

JDO metadata:

```

<class name="Magazine" objectid-class="Magazine$MagazineID">
  <field name="id" primary-key="true"/>
  ... field metadata ...
</class>

```

Mapping information using the mapping XML format:

```

<class name="Magazine">
  <jdbc-class-map type="horizontal"/>
</class>

```

Mapping information using JDO metadata extensions:

```
<class name="Magazine" objectid-class="Magazine$MagazineID">
  <extension vendor-name="kodo" key="jdbc-class-map" value="horizontal"/>
  <field name="id" primary-key="true"/>
  ... field metadata ...
</class>
```

Java class:

```
public class Tabloid
    extends Magazine
{
    private String tabloidType;
    ... class content ...

    public static class TabloidID
        extends Magazine.MagazineID
    {
        public TabloidID ()
        {
            super ();
        }

        public TabloidID (String idString)
        {
            super (idString);
        }

        // note that equals() and hashCode() methods in superclass
        // are sufficient for application identity hierarchy
    }
}
```

Schema:

```
<table name="TABLOID">
  <pk column="ID"/>
  <column name="ID" type="bigint"/>
  <column name="TABLOID_TYPE" type="varchar" size="255"/>
  <column name="TITLE" type="varchar" size="255"/>
  ... columns for other tabloid fields ...
</table>
```

JDO metadata:

```
<class name="Tabloid" persistence-capable-superclass="Magazine"
  objectid-class="Tabloid$TabloidID">
  ... field metadata ...
</class>
```

Mapping information using the mapping XML format:

```
<class name="Tabloid">
  <jdbc-class-map type="base" table="TABLOID"/>
  ... indicator mappings ...
  <field name="Magazine.id">
    <jdbc-field-map type="value" column="ID"/>
  </field>
  <field name="tabloidType">
    <jdbc-field-map type="value" column="TABLOID_TYPE"/>
  </field>
  <field name="Magazine.title">
    <jdbc-field-map type="value" column="TITLE"/>
  </field>
  ... other field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Tabloid" persistence-capable-superclass="Magazine"
  objectid-class="Tabloid$TabloidID">
  <extension vendor-name="kodo" key="jdbc-class-map" value="base">
    <extension vendor-name="kodo" key="table" value="TABLOID"/>
  </extension>
  ... indicator extensions ...
  <field name="tabloidType">
    <extension vendor-name="kodo" key="jdbc-field-map" value="value">
      <extension vendor-name="kodo" key="column" value="TABLOID_TYPE"/>
    </extension>
  </field>
```

```
<extension vendor-name="kodo" key="jdbc-field-mappings">
  <extension vendor-name="kodo" key="Magazine.id">
    <extension vendor-name="kodo" key="jdbc-field-map" value="value">
      <extension vendor-name="kodo" key="column" value="ID"/>
    </extension>
  </extension>
  <extension vendor-name="kodo" key="Magazine.title">
    <extension vendor-name="kodo" key="jdbc-field-map" value="value">
      <extension vendor-name="kodo" key="column" value="TITLE"/>
    </extension>
  </extension>
</extension>
</class>
```

### 7.6.4.1. Special considerations when using Horizontal Inheritance Mapping

---

- *Declaring classes abstract*: Kodo does not require that a class that is horizontally mapped be declared abstract. However, it is recommended that these classes not be concrete, since any attempt to persist a superclass that is horizontally mapped will throw a `FatalUserException`.
- *Application identity*: When a topmost horizontally-mapped superclass uses application identity, one of the following two restrictions must be observed by the application developer:
  1. The primary key values in each of the tables that extend the horizontally mapped superclass must be unique. That is, if `MySuperClass` declares horizontal mapping, and `MySubClass1` and `MySubClass2` extend `MySuperClass` and are mapped to "MY\_SUB1" and "MY\_SUB2" (respectively), then there can be no row in MY\_SUB1 that has a primary key value that exists in MY\_SUB2. This is because a call to `getObjectById` with an application identity instance cannot identify which subclass the ID is associated with.
  2. Rather than having a single application identity class associated with the entire class hierarchy, each subclass declares its own application identity class. The inheritance hierarchy of the application identity classes must exactly match the inheritance hierarchy of the persistent classes they represent. This enables Kodo to unambiguously identify which subclass a specific application identity instance represents.
- *Relations*: Relations to horizontally mapped classes will be treated the same as relations to interfaces or fields of type `PersistenceCapable`. That is, the relations will be stored as the stringified object id.

### 7.6.4.2. Advantages of using Horizontal Inheritance Mapping

---

An advantage of using a horizontal mapping is that attributes that are common to multiple persistent classes can be defined in the superclass without having to suffer the performance consequences and relational design restrictions of using a vertical mapping. Persisting and modifying instances of subclasses whose superclass uses horizontal mapping is as fast as flat inheritance mapping, since typically only a single `INSERT` or `UPDATE` statement will need to be issued.

### 7.6.4.3. Disadvantages of using Horizontal Inheritance Mapping

---

In addition to the restrictions imposed by using horizontal mappings (see [Section 7.6.4.1, “Special considerations when using Horizontal Inheritance Mapping”](#) [228]), queries against the superclass fields of a horizontally-mapped class will require multiple `SELECT` statements to be executed against the database (one for each concrete subclass). Furthermore, from a relational design standpoint, horizontal mappings are not normalized, since attributes are repeated across different tables.

## 7.6.5. Custom Class Mapping

---

Kodo JDO allows you to create your own class mappings. A custom class mapping can override any or all of the CRUD operations for objects of the class: Create, Retrieve, Update, Delete. All mappings must extend, directly or indirectly, from `kodo.jdbc.meta.ClassMapping`.

The `jdbc-class-map-name` JDO metadata extension tells the **mapping tool** which class mapping to install. If you write mappings by hand rather than with the mapping tool, simply specify the full class name of your custom mapping in the `type` attribute of the mapping XML.

The `samples/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.

### Note

Kodo's custom class mapping capabilities require a Kodo JDO Enterprise Edition license in order to function. See [Chapter 13, Enterprise Edition \[321\]](#) for details about the Enterprise Edition.

## 7.7. Version Indicator

A *version indicator* is responsible for versioning each stored object so that optimistic locking errors can be detected. The version indicator is always placed on the base class in an inheritance tree. Subclasses inherit the version indicator of their parent class.

In Kodo JDO, version indicators extend the base `kodo.jdbc.meta.VersionIndicator` class. The concrete version indicators Kodo JDO provides are described in the sections below. By default, the **mapping tool** uses the **version-number** version indicator for all persistent classes. You can change the default version indicator type with the `kodo.jdbc.VersionIndicator` configuration property. You can also instruct the mapping tool to use a specific indicator for an individual class with the `jdbc-version-ind-name` JDO metadata extension.

Both the number and date version indicators can be used in conjunction with Kodo's lock group support. Details regarding use and limitations of field-level lock groups are available in [Section 14.6, “Lock Groups” \[340\]](#)

### 7.7.1. Version Number Indicator

The version number indicator uses a version number stored in a database column to detect concurrent modifications to an object. When an object is loaded, its version number is loaded along with it. On transaction commit, the in-memory version number is compared to the database version number. If the database version number does not match, then the object has been modified concurrently by another transaction, and an optimistic lock exception is raised. Otherwise, the version number is incremented as a sign to other transactions that the object was changed.

The version number indicator has the following attributes:

- `type`: version-number
- `column`: The name of the column that holds the version number for each object. This column must be in the class' primary table. This property is required.

#### *Example 7.21. Using a Version Number Indicator*

```
Java class:
public class Magazine
{
    ... class content ...
}
```

Schema:

```
<table name="MAGAZINE">
  ... primary key columns ...
  <column name="JDOVERSION" type="bigint"/>
  ... columns for magazine fields ...
</table>

JDO metadata:

<class name="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping ...
  <jdbc-version-ind type="version-number" column="JDOVERSION"/>
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions ...
  <extension vendor-name="kodo" key="jdbc-version-ind" value="version-number">
    <extension vendor-name="kodo" key="column" value="JDOVERSION"/>
  </extension>
  ... field metadata ...
</class>
```

### Note

When adding a version column to an existing schema, the values of the version should be seeded with zero, rather than null. Kodo expects non-null values for the contents of a version column.

## 7.7.2. Version Date Indicator

---

The version date indicator uses a timestamp column to track the last revision to an object. The timestamp is updated whenever the object's database record is updated. On transaction commit, timestamps are compared to be sure that the object wasn't changed by another transaction since it was loaded by the current one.

The version date indicator is intended for legacy support only. The **version number** indicator is both more efficient and more robust, because it is not dependent on the granularity of the database's timestamps.

The version date indicator has the following attributes:

- `type`: version-date
- `column`: The name of the column that holds the SQL timestamp for each object. This column must be in the class' primary table. This property is required.

### *Example 7.22. Using a Version Date Indicator*

```
Java class:

public class Magazine
{
  ... class content ...
}
```

```
Schema:
<table name="MAGAZINE">
  ... primary key columns
  <column name="JDOVERSION" type="timestamp"/>
  ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping ...
  <jdbc-version-ind type="version-date" column="JDOVERSION"/>
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
  ... class extensions ...
  <extension vendor-name="kodo" key="jdbc-version-ind" value="version-date">
    <extension vendor-name="kodo" key="column" value="JDOVERSION"/>
  </extension>
  ... field metadata ...
</class>
```

---

### 7.7.3. State Image Indicator

The state image indicator does not require any database columns in order to detect optimistic lock violations. Instead, it maintains an image of the object's state as it is loaded. When the object is committed for update, the indicator checks the in-memory image against the database values to make sure no other transaction has changed the object's record since it was loaded. The following limitations apply to the state image indicator:

- State comparisons only use simple value fields such as strings, primitives, and primitive wrappers.
- Comparisons exclude doubles, floats, and dates because they cannot be compared reliably.
- If concurrent transactions modify fields in a disjoint set of tables, the conflict will go undetected.
- Fields with the "none" lock group are not included in state image checks and conflict will go undetected.

The state image indicator has the following attributes:

- type: state-image

#### *Example 7.23. Using a State Image Indicator*

```
Java class:
public class Magazine
{
  ... class content ...
}
```

```
Schema:
<table name="MAGAZINE">
  ... primary key columns ...
  ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping ...
  <jdbc-version-ind type="state-image"/>
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
  ... class extensions ...
  <extension vendor-name="kodo" key="jdbc-version-ind" value="state-image"/>
  ... field metadata ...
</class>
```

---

## 7.7.4. Custom Version Indicator

Kodo JDO allows you to create your own version indicators. All version indicators must extend, directly or indirectly, from `kodo.jdbc.meta.VersionIndicator`.

The `jdbc-version-ind-name` JDO metadata extension tells the **mapping tool** which version indicator to install. If you write mappings by hand rather than with the mapping tool, simply specify the full class name of your custom indicator in the `type` attribute of the mapping XML.

The `samples/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.

---

## 7.8. Class Indicator

A *class indicator* determines what class of object each database record represents. A class indicator is only necessary when a persistent class can be extended by other persistent classes. The class indicator is always placed on the base class. Subclasses inherit the class indicator of their parent class.

In Kodo JDO, class indicators extend the base `kodo.impl.meta.ClassIndicator` class. The concrete class indicators Kodo JDO provides are described in the sections below. By default, the **mapping tool** uses the `in-class-name` class indicator for all persistent classes. You can change the default class indicator type with the `kodo.jdbc.ClassIndicator` configuration property. You can also instruct the mapping tool to use a specific indicator for an individual class with the `jdbc-class-ind-name` JDO metadata extension.

---

### 7.8.1. In-Class-Name Indicator

The `in-class-name` indicator stores the full class name of persistent objects in a special database column. When you query for objects of certain classes, the indicator appends a SQL IN clause to the end of the query to make sure each record's class name column matches one of target class names.

The `in-class-name` indicator has the following attributes:

- `type`: in-class-name
- `column`: The name of the column that stores the full class name of each persistent object. The column must be in the class' primary table. This property is required.

### *Example 7.24. Using an In-Class-Name Indicator*

```
Java class:
public class Magazine
{
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns ...
    <column name="JDOCLASS" type="varchar" size="255"/>
    ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    ... class mapping ...
    <jdbc-class-ind type="in-class-name" column="JDOCLASS"/>
    ... field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
    ... class extensions ...
    <extension vendor-name="kodo" key="jdbc-class-ind" value="in-class-name">
        <extension vendor-name="kodo" key="column" value="JDOCLASS"/>
    </extension>
    ... field metadata ...
</class>
```

---

## 7.8.2. Metadata Value Indicator

The metadata-value indicator maps persistent classes to symbolic constants stored in a database column. Each class in the inheritance tree using the metadata value indicator uses the **jdbc-class-ind-value** metadata extension to specify the database value that indicates a row is a member of that class. The extension is required. When you query for objects of certain classes, the indicator appends a SQL IN clause to the end of the query to make sure each record's class indicator column value matches the symbolic constant mapped to one of those classes.

Unlike the **in-class-name** indicator, this indicator cannot calculate all the subclasses for a given class by examining the database contents. If you use this indicator, you must list all of your persistent classes in the **kodo.PersistentClasses** property, or make sure all classes in inheritance trees that use this indicator have been referenced in code by the time you perform persistent operations on any of them.

The metadata-value indicator has the following attributes:

- type: metadata-value
- column: The name of the column that stores the class indicator value for each row. The column must be in the class' primary table. This property is required.

### ***Example 7.25. Using a Metadata-Value Indicator***

```
Java class:

public abstract class Person
{
    ... class content ...
}

public class Male
    extends Person
{
    ... class content ...
}

public class Female
    extends Person
{
    ... class content ...
}

Schema:

<table name="PERSON">
    ... primary key columns ...
    <column name="SEX" type="char" size="1"/>
    ... columns for person fields ...
</table>

JDO metadata:

<class name="Person">
    <!-- person is abstract, so no class-ind-value required -->
    ... field metadata ...
</class>

<class name="Male" persistence-capable-superclass="Person">
    <extension vendor-name="kodo" key="jdbc-class-ind-value" value="M"/>
    ... field metadata ...
</class>

<class name="Female" persistence-capable-superclass="Person">
    <extension vendor-name="kodo" key="jdbc-class-ind-value" value="F"/>
    ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Person">
    ... class mapping ...
    <jdbc-class-ind type="metadata-value" column="SEX"/>
    ... field mappings ...
</class>

<class name="Male">
    ... class mapping ...
    ... field mappings ...
</class>

<class name="Female">
    ... class mapping ...
    ... field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Person">
    ... class extensions ...
    <extension vendor-name="kodo" key="jdbc-class-ind" value="metadata-value">
        <extension vendor-name="kodo" key="column" value="SEX"/>
    </extension>
    ... field metadata ...
</class>

<class name="Male" persistence-capable-superclass="Person">
```

```
... class extensions ...
<extension vendor-name="kodo" key="jdbc-class-ind-value" value="M"/>
... field metadata ...
</class>

<class name="Female" persistence-capable-superclass="Person">
... class extensions ...
<extension vendor-name="kodo" key="jdbc-class-ind-value" value="F"/>
... field metadata ...
</class>
```

### 7.8.3. Subclass-Join Indicator

---

The subclass-join indicator determines the class or each database record using outer joins to subclass tables. It does not require a special column to store class information. You can only use this indicator if your database supports outer joins, and if every subclass in your inheritance hierarchy is vertically mapped (see [Section 7.6.3, “Vertical Inheritance Mapping” \[222\]](#)).

The subclass-join indicator has the following attributes:

- type: subclass-join

#### *Example 7.26. Using a Subclass-Join Indicator*

```
Java class:

public class Magazine
{
    ... class content ...
}

public class Tabloid
{
    extends Magazine
    {
        ... class content ...
    }
}

Schema:

<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="TABLOID">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  ... columns for tabloid fields ...
</table>

JDO metadata:

<class name="Magazine">
  ... field metadata ...
</class>

<class name="Tabloid" persistence-capable-superclass="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping ...
  <jdbc-class-ind type="subclass-join"/>
  ... field mappings ...
</class>
```

```
<class name="Tabloid">
  <jdbc-class-map type="vertical" table="TABLOID" ref-column.JDOID="MAG_ID"/>
  ... indicator mappings ...
  ... field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Magazine">
  ... class extensions ...
  <extension vendor-name="kodo" key="jdbc-class-ind" value="subclass-join"/>
  ... field metadata ...
</class>

<class name="Tabloid" persistence-capable-superclass="Magazine">
  <extension vendor-name="kodo" key="jdbc-class-map" value="vertical">
    <extension vendor-name="kodo" key="table" value="TABLOID"/>
    <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
  </extension>
  ... indicator extensions ...
  ... field metadata ...
</class>
```

---

## 7.8.4. Custom Class Indicator

Kodo JDO allows you to create your own class indicators. All class indicators must extend, directly or indirectly, from `kodo.jdbc.meta.ClassIndicator`.

The `jdbc-class-ind-name` JDO metadata extension tells the **mapping tool** which class indicator to install. If you write mappings by hand rather than with the mapping tool, simply specify the full class name of your custom indicator in the `type` attribute of the mapping XML.

The `samples/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.

---

## 7.9. Field Mapping

A *field mapping* describes how a persistent field maps to the database. If necessary, it also contains data on how to link the field to the data of its owning object, and how to link the field to the data of any related objects (if the field represents a relation to one or more other persistent objects).

In Kodo JDO, field mappings extend the base `kodo.jdbc.meta.FieldMapping` class. The concrete field mappings Kodo JDO provides are described in the sections below. By default, the **mapping tool** decides which field mapping to use for a particular field by cycling through all the known mappings until one returns `true` from its `map` method. You can instruct the mapping tool to use a specific mapping for an individual field with the `jdbc-field-map-name` JDO metadata extension.

This section concludes with a discussion of externalization in [Section 7.9.23, “Externalization” \[273\]](#), a Kodo feature that allows you to persist many field types that aren't supported directly by JDO without a custom field mapping, and without resorting to serialization.

---

### 7.9.1. Value Mapping

The value mapping represents the direct mapping of a Java primitive, primitive wrapper, `Date`, or string to a compatibly-typed database column. The value mapping has the following attributes:

- `type`: value
- `column`: The name of the column that stores the field value. This property is required.

- `table`: If the column is not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

### *Example 7.27. Using a Value Mapping*

```
Java class:
public class Magazine
{
    private String isbn;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns ...
    <column name="ISBN" type="varchar" size="10"/>
    ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    ... class mapping ...
    ... indicator mappings ...
    <field name="isbn">
        <jdbc-field-map type="value" column="ISBN"/>
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
    ... class extensions ...
    ... indicator extensions ...
    <field name="isbn">
        <extension vendor-name="kodo" key="jdbc-field-map" value="value">
            <extension vendor-name="kodo" key="column" value="ISBN"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>
```

**Example 7.28. Using a Value Mapping in a Separate Table**

Note that though we do not include separate-table examples for any other field mappings, all other built-in field mapping types can be used in a separate table, following the same pattern as the example below.

```

Java class:

public class Magazine
{
    private String isbn;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
  <column name="JDROID" type="BIGINT"/>
  <pk column="JDROID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_EXTRAS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDROID"/>
  </fk>
  <column name="ISBN" type="varchar" size="10"/>
  ... columns for other fields ...
</table>

JDO metadata:

<class name="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping
  ... indicator mappings
  <field name="isbn">
    <jdbc-field-map type="value" column="ISBN"
      table="MAG_EXTRAS" ref-column.JDROID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions
  ... indicator extensions
  <field name="isbn">
    <extension vendor-name="kodo" key="jdbc-field-map" value="value">
      <extension vendor-name="kodo" key="column" value="ISBN"/>
      <extension vendor-name="kodo" key="table" value="MAG_EXTRAS"/>
      <extension vendor-name="kodo" key="ref-column.JDROID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.2. Blob Mapping

The blob mapping serializes the value of the Java field it is installed on, and stores the serialized bytes in a binary database column. Fields whose type is unrecognized or `java.lang.Object`, collections and arrays whose `element-type` is unspecified, unrecognized, or set to `java.lang.Object`, maps whose `key-type`, `value-type`, or both is unspecified, unrecognized, or `java.lang.Object`, and non-string fields that have set the **`jdbc-size`** JDO metadata extension to -1 all default to using blob mappings. Also, see the **`type`** JDO metadata extension for how to force an interface-typed field to use a blob

mapping.

The blob mapping has the following attributes:

- `type`: blob
- `column`: The name of the binary column that stores the field value. This property is required.
- `table`: If the column is not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

### *Example 7.29. Using a Blob Mapping*

```
Java class:
public class Magazine
{
    private Object data;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns
    <column name="DATA" type="blob"/>
    ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
    <field name="data" persistence-modifier="persistent"/>
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    ... class mapping
    ... indicator mappings
    <field name="data">
        <jdbc-field-map type="blob" column="DATA"/>
    </field>
    ... rest of field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Magazine">
  ... class extensions
  ... indicator extensions
  <field name="data">
    <extension vendor-name="kodo" key="jdbc-field-map" value="blob">
      <extension vendor-name="kodo" key="column" value="DATA"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>
```

### 7.9.3. Clob Mapping

The clob mapping is reserved for string fields that map to a database CLOB column. You can indicate that a string should be stored as a CLOB by setting the **jdbc-size** JDO metadata extension on the field to -1. Note that some databases can support string of unlimited length without using a CLOB; when this is the case the **mapping tool** will install a **value mapping** in favor of this mapping.

The clob mapping has the following attributes:

- **type**: clob
- **column**: The name of the CLOB column that stores the field value. This property is required.
- **table**: If the column is not in the table listed by the owning class mapping, specify the table name here.
- **ref-column**.**<pk column>\***: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each **ref-column** attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- **ref-constant**.**<column>\***: Similar to the **ref-column** attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- **ref-join-type**: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to **outer**, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

#### *Example 7.30. Using a Clob Mapping*

```
Java class:
public class Magazine
{
  private String coverStorySummary;
  ... class content ...
}
```

```
Schema:
<table name="MAGAZINE">
... primary key columns ...
<column name="COVERSUMMARY" type="clob"/>
... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
  <field name="coverStorySummary">
    <extension vendor-name="kodo" key="jdbc-size" value="-1"/>
  </field>
  ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
... class mapping ...
... indicator mappings ...
  <field name="coverStorySummary">
    <jdbc-field-map type="clob" column="COVERSUMMARY"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
... class extensions ...
... indicator extensions ...
  <field name="coverStorySummary">
    <extension vendor-name="kodo" key="jdbc-size" value="-1"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="clob">
      <extension vendor-name="kodo" key="column" value="COVERSUMMARY"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>
```

---

## 7.9.4. Byte Array Mapping

The byte array mapping is a mapping specifically for byte array fields. Unlike **blob mapping**, this mapping does not serialize or deserialize your byte array and instead stores the array directly to the database. This is useful for binary columns which are generated by legacy or non-Java applications.

The byte array mapping has the following attributes:

- `type`: byte-array
- `column`: The name of the binary column that stores the field value. This property is required.
- `table`: If the column is not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see [Section 7.5.2, “Non-Standard Joins” \[218\]](#).
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table,

this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

### ***Example 7.31. Using a Byte array Mapping***

```
Java class:
public class Magazine
{
    private byte [] coverData;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns
    <column name="COVER_DATA" type="blob"/>
    ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    ... class mapping
    ... indicator mappings
    <field name="coverData">
        <jdbc-field-map type="byte-array" column="COVER_DATA"/>
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
    ... class extensions
    ... indicator extensions
    <field name="data">
        <extension vendor-name="kodo" key="jdbc-field-map" value="byte-array">
            <extension vendor-name="kodo" key="column" value="COVER_DATA"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>
```

---

## **7.9.5. One-to-One Mapping**

A one-to-one mapping represents a field that holds a reference to another persistent object. One-to-one mappings can be one-sided or two-sided. In the latter case, two objects hold a reference to each other, but only the database record for the "owner" of the relation holds the primary key values of the inverse object.

If your database defines a non-deferred foreign key constraint for the relation, you should be sure to set the

`kodo.jdbc.ForeignKeyConstraints` property appropriately.

The one-to-one mapping has the following attributes:

- `type: one-one`
- `column.<pk column>*`: Kodo JDO must be able to join this mapping's columns to the row of the related object. Each `column` attribute joins a column in the local table to the corresponding column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- `table`: If the join columns are not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the join columns are not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

- `class-criteria`: Whether or not to use the expected class of the related object as a criteria in the SQL issued to select its database record. Typically, this is not needed; Kodo can just use the foreign key value to select the desired record, and if your database data is valid, the record will correspond to an object of the correct class. Under some very rare mappings, however, you may need to select based on both foreign key and class -- for example, if you are using an inverse-based one-one relation that shares the inverse foreign key with another inverse-based one-one to an object of a different subclass. In these rare cases, set this attribute to `true` to force Kodo to append class conditions to its SQL.

### *Example 7.32. Using a One-to-One Mapping*

```
Java class:
public class Magazine
{
    private Headquarters hq;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns ...
    <column name="HQID" type="bigint"/>
    <fk to-table="HEADQUARTERS">
        <join column="HQID" to-column="ID"/>
    </fk>
    ... columns for magazine fields ...
</table>

<table name="HEADQUARTERS">
    <column name="ID" type="bigint"/>
    <pk column="ID"/>
    ... columns for headquarters fields ...
```

```

</table>

JDO metadata:

<class name="Magazine">
  ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="hq">
    <jdbc-field-map type="one-one" column.ID="HQID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions ...
  <field name="hq">
    <extension vendor-name="kodo" key="jdbc-field-map" value="one-one">
      <extension vendor-name="kodo" key="column.ID" value="HQID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

### Example 7.33. Using a Two-Sided One-to-One Mapping

This example is the same as the previous one, except that now the Headquarters class includes an inverse relation to Magazine. The MAGAZINE table columns for the hq field control both sides of the relation; the HEADQUARTERS table does not hold any record of the owning Magazine. If this were not the case and the HEADQUARTERS table held an independent reference to the owning Magazine then this would not be a two-sided mapping in Kodo JDO; it would simply be two separate, regular one-to-one mappings through the Magazine.hq and Headquarters.mag fields.

Notice the use of the **inverse-owner** JDO metadata extension to mark the "owning" side of the relation. Also note that Headquarters.mag field simply links the MAGAZINE table primary key columns right back to themselves in its column attributes, while using the MAGAZINE table's foreign key back to HEADQUARTERS as its ref-column joins.

```

Java class:

public class Magazine
{
  private Headquarters hq;
  ... class content ...
}

public class Headquarters
{
  private Magazine mag;
  ... class content ...
}

Schema:

<table name="MAGAZINE">
  <column name="JDOID" type="bigint" not-null="true"/>
  <pk column="JDOID"/>
  <column name="HQID" type="bigint"/>
  <fk to-table="HEADQUARTERS">
    <join column="HQID" to-column="ID"/>
  </fk>
  ... columns for magazine fields ...
</table>

<table name="HEADQUARTERS">
  <column name="ID" type="bigint"/>

```

```

    <pk column="ID" />
    ... columns for headquarters fields ...
</table>

JDO metadata:

<class name="Magazine">
    ... field metadata ...
</class>

<class name="Headquarters">
    <field name="mag">
        <extension vendor-name="kodo" key="inverse-owner" value="hq" />
    </field>
    ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
    ... class mapping
    ... indicator mappings
    <field name="hq">
        <jdbc-field-map type="one-one" column.ID="HQID" />
    </field>
    ... rest of field mappings ...
</class>

<class name="Headquarters">
    ... class mapping
    ... indicator mappings
    <field name="mag">
        <jdbc-field-map type="one-one" table="MAGAZINE" ref-column.ID="HQID"
            column.JDOID="JDOID" />
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
    ... class extensions
    ... indicator extensions
    <field name="hq">
        <extension vendor-name="kodo" key="jdbc-field-map" value="one-one">
            <extension vendor-name="kodo" key="column.ID" value="HQID">
            </extension>
        </extension>
    </field>
    ... rest of field metadata ...
</class>

<class name="Headquarters">
    ... class extensions
    ... indicator extensions
    <field name="mag">
        <extension vendor-name="kodo" key="inverse-owner" value="hq">
        <extension vendor-name="kodo" key="jdbc-field-map" value="one-one">
            <extension vendor-name="kodo" key="table" value="MAGAZINE" />
            <extension vendor-name="kodo" key="ref-column.ID" value="HQID" />
            <extension vendor-name="kodo" key="column.JDOID" value="JDOID" />
        </extension>
        </field>
    ... rest of field metadata ...
</class>

```

### ***Example 7.34. Using a One-to-One Mapping With Inverse Columns***

In this example, the `Magazine` class no longer has a relation to `Headquarters`. However, the `MAGAZINE` table still has the inverse columns that hold the primary key values of the related `Headquarters` instances. The `Headquarters.mag` field uses these inverse columns. Notice that the mapping data for `Headquarters.mag` is the same in this example as in the previous one; the only difference is the absence of the `inverse-owner` JDO metadata extension.

```

Java class:

public class Headquarters

```

```

{
    private Magazine mag;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
    <column name="JDOID" type="bigint" not-null="true"/>
    <pk column="JDOID"/>
    <column name="HQID" type="bigint"/>
    <fk to-table="HEADQUARTERS">
        <join column="HQID" to-column="ID"/>
    </fk>
    ... columns for magazine fields ...
</table>

<table name="HEADQUARTERS">
    <column name="ID" type="bigint"/>
    <pk column="ID"/>
    ... columns for headquarters fields ...
</table>

JDO metadata:

<class name="Headquarters">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Headquarters">
    ... class mapping ...
    ... indicator mappings ...
    <field name="mag">
        <jdbc-field-map type="one-one" table="MAGAZINE" ref-column.ID="HQID"
            column.JDOID="JDOID"/>
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Headquarters">
    ... class extensions ...
    ... indicator extensions ...
    <field name="mag">
        <extension vendor-name="kodo" key="jdbc-field-map" value="one-one">
            <extension vendor-name="kodo" key="table" value="MAGAZINE"/>
            <extension vendor-name="kodo" key="ref-column.ID" value="HQID"/>
            <extension vendor-name="kodo" key="column.JDOID" value="JDOID"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>

```

## 7.9.6. PC One-to-One Mapping

The PC one-to-one mapping is used when a field holds a persistence-capable object of unknown type. It is the default mapping for user-defined interface fields, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), and fields whose **type** JDO metadata extension is set to `PersistenceCapable`. The mapping uses a single column to hold the stringified value of the related persistent object's oid value. You cannot query across this mapping.

The PC one-to-one mapping has the following attributes:

- **type**: pc
- **column**: The name of the string-compatible column that stores the stringified oid of the related object. This property is required.
- **table**: If the column is not in the table listed by the owning class mapping, specify the table name here.

- `ref-column.<pk column>*`: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

### *Example 7.35. Using a PC One-to-One Mapping*

```
Java class:
public class Magazine
{
    private IFormat format;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    ... primary key columns
    <column name="FORMAT" type="varchar" size="255"/>
    ... columns for magazine fields ...
</table>

JDO metadata:
<class name="Magazine">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
    ... class mapping
    ... indicator mappings
    <field name="format">
        <jdbc-field-map type="pc" column="FORMAT"/>
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
    ... class extensions
    ... indicator extensions
    <field name="hq">
        <extension vendor-name="kodo" key="jdbc-field-map" value="pc">
            <extension vendor-name="kodo" key="column" value="FORMAT"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>
```

## 7.9.7. Embedded One-to-One Mapping

The embedded one-to-one mapping places the columns needed for the fields of a related object within the owning class' table. This mapping is often used for simple, privately-owned relations, such as a `User` object's relation to its unique `Address`.

The behavior of embedded objects in JDO is different than that of other persistent objects. First, embedded objects do not appear in JDO extents or query results. Also, embedded objects are not shared among owners. If two `Users` reference the same `Address` on commit, each `User` will have a distinct copy of the `Address` when the commit process concludes. You do not call `makePersistent` on embedded objects. Doing so will create a separate, independent instance in the table for the object's class. Simply assigning an object to an embedded field will ensure that it is made persistent and embedded on transaction commit. And finally, when you null an embedded field or assign a new value to the field, it is as if you deleted the old value.

Kodo JDO places very few restrictions on embedded objects. Embedded objects can have all the same field types as other persistent objects, including recursively embedded fields of their own. Classes can have instances that are independently persistent and other instances that are embedded. If you have a class whose instances are always embedded in other objects and never persisted to their own table, you should set the class' `jdbc-class-map-name` JDO metadata extension to `none` to prevent the **mapping tool** from creating a mapping and table for the class.

There are only two restrictions on embedded fields: an embedded field's declared type must exactly match the type of the embedded value (the embedded object cannot be a subclass of the field's declared type), and fields cannot be embedded in such a way that infinite recursion results. So, for example, a `User` cannot embed another field of type `User`.

Embedded mappings have the properties bulleted below. They also have nested `field` elements, as you will see in the upcoming example.

- `type: embedded`
- `null-ind-column`: The name of the column that differentiates between a null embedded object and an embedded object that happens to have default values for all of its fields. This property is required. The embedded object will be null if the column's value is null. By default, the mapping tool adds a synthetic null indicator column when creating an embedded mapping. You can tell the mapping tool which embedded field's column to use as the null indicator column through the `jdbc-null-ind-name` JDO metadata extension.
- `synthetic`: This property is `true` if the null indicator column is a synthetic column for the sole purpose of determining whether the embedded object is null, or `false` if the null indicator column actually holds data for one of the embedded object's fields.
- `table`: If the embedded object's data is not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the embedded object's data is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.

### Example 7.36. Using an Embedded One-to-One Mapping

Notice that when you map an embedded object, you map all of its fields using nested `field` elements. This process can be recursive.

```
Java class:
public class Magazine
{
```

```

    private ContactInfo contact;
    ... class content ...
}

public class ContactInfo
{
    private String email;
    private String phone;
    private String fax;
    ... class content ...
}

```

Schema:

```

<table name="MAGAZINE">
    ... primary key columns ...
    <column name="CONTACT_EMAIL" type="varchar" size="255"/>
    <column name="CONTACT_PHONE" type="varchar" size="15"/>
    <column name="CONTACT_FAX" type="varchar" size="15"/>
    ... columns for magazine fields ...
</table>

```

JDO metadata:

```

<class name="Magazine">
    <field name="contact" embedded="true"/>
    ... rest of field metadata ...
</class>
<class name="ContactInfo"/>

```

Mapping information using the mapping XML format:

```

<class name="Magazine">
    ... class mapping ...
    ... indicator mappings ...
    <field name="contact">
        <jdbc-field-map type="embedded" null-ind-column="CONTACT_EMAIL"
            synthetic="false">
            <field name="email">
                <jdbc-field-map type="value" column="CONTACT_EMAIL"/>
            </field>
            <field name="phone">
                <jdbc-field-map type="value" column="CONTACT_PHONE"/>
            </field>
            <field name="fax">
                <jdbc-field-map type="value" column="CONTACT_FAX"/>
            </field>
        </jdbc-field-map>
    </field>
    ... rest of field mappings ...
</class>
<class name="ContactInfo">
    <jdbc-class-map type="none"/>
    <field name="email">
        <jdbc-field-map type="none"/>
    </field>
    <field name="phone">
        <jdbc-field-map type="none"/>
    </field>
    <field name="fax">
        <jdbc-field-map type="none"/>
    </field>
</class>

```

Mapping information using JDO metadata extensions:

```

<class name="Magazine">
    ... class extensions ...
    ... indicator extensions ...
    <field name="contact" embedded="true">
        <extension vendor-name="kodo" key="jdbc-field-map" value="embedded">
            <extension vendor-name="kodo" key="null-ind-column" value="CONTACT_EMAIL"/>
            <extension vendor-name="kodo" key="synthetic" value="false"/>
            <extension vendor-name="kodo" key="email">
                <extension vendor-name="kodo" key="jdbc-field-map" value="value">
                    <extension vendor-name="kodo" key="column" value="CONTACT_EMAIL"/>
                </extension>
            </extension>
            <extension vendor-name="kodo" key="phone">
                <extension vendor-name="kodo" key="jdbc-field-map" value="value">
                    <extension vendor-name="kodo" key="column" value="CONTACT_PHONE"/>
                </extension>
            </extension>
            <extension vendor-name="kodo" key="fax">
                <extension vendor-name="kodo" key="jdbc-field-map" value="value">
                    <extension vendor-name="kodo" key="column" value="CONTACT_FAX"/>
                </extension>
            </extension>
        </extension>
    </field>
</class>

```

```

    </field>
    ... rest of field metadata ...
</class>
<class name="ContactInfo">
  <extension vendor-name="kodo" key="jdbc-class-map" value="none"/>
  <field name="email">
    <extension vendor-name="kodo" key="jdbc-field-map" value="none"/>
  </field>
  <field name="phone">
    <extension vendor-name="kodo" key="jdbc-field-map" value="none"/>
  </field>
  <field name="fax">
    <extension vendor-name="kodo" key="jdbc-field-map" value="none"/>
  </field>
</class>

```

## 7.9.8. Enumeration Mapping

The enum mapping represents the mapping of a Java 5 enum type to a `String`-compatible database column. This mapping stores the enum's name, not its ordinal position. The enum mapping has the following attributes:

- `type`: enum
- `column`: The name of the column that stores the enum name. This property is required.
- `table`: If the column is not in the table listed by the owning class mapping, specify the table name here.
- `ref-column.<pk column>*`: If the column is not in the table listed by the owning class mapping, Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `ref-join-type`: By default, Kodo assumes that when a field value resides in a table other than the class' primary table, this other table always contains a row for each primary table row. Thus Kodo uses a standard inner join when selecting for other-table values, and when inserting a new object Kodo always inserts a row into the other table as well.

If you set this attribute to `outer`, however, Kodo will not assume that there is a row in this field's table for each row in the class' primary table. Kodo will use outer joins to retrieve the values of this field. Furthermore, Kodo will not insert a row into this field's table unless the field is non-null. When updating an object, the field's row will be deleted if the field value has been set to null (or the equivalent Java default for primitives). Note that Kodo also takes into account other fields that might share a row with this one, so that, for example, the row is only deleted if all fields sharing this table are null for the object in question.

### *Example 7.37. Using an Enum Mapping*

```

Java class:
public class Subscription
{
    public enum SubscriptionType
    {
        EDUCATIONAL,
        RESIDENTIAL
    }

    private SubscriptionType type;
    ... class content ...
}

```

```
Schema:
<table name="SUBSCRIPTION">
  ... primary key columns
  <column name="TYPE" type="varchar" size="11"/>
  ... columns for subscription fields ...
</table>

JDO metadata:
<class name="Subscription">
  ... field metadata ...

  <!-- Since enums are not yet included in the JDO spec as types
        that should be persistent by default, you must identify
        enum fields that should be persisted. This behavior may
        change in a future version of Kodo, in accordance with the
        JDO2 specification. -->
  <field name="type" persistence-modifier="persistent"/>
</class>

Mapping information using the mapping XML format:
<class name="Subscription">
  ... class mapping
  ... indicator mappings
  <field name="type">
    <jdbc-field-map type="enum" column="TYPE"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Subscription">
  ... class extensions
  ... indicator extensions
  <field name="type">
    <extension vendor-name="kodo" key="jdbc-field-map" value="enum">
      <extension vendor-name="kodo" key="column" value="TYPE"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>
```

---

## 7.9.9. Collection Mapping

The collection mapping maps a collection or array of simple values (primitive wrappers, dates, or strings). It has the following attributes:

- `type`: collection
- `table`: The table where the collection elements are stored. This property is required.
- `element-column`: The column that holds each element value. This property is required.
- `order-column`: The column that holds the position of each element within its collection. This property is optional. By default, the **mapping tool** orders fields of list or array types, and leaves collection and set fields unordered. You can control whether the mapping tool adds an order column with the **`jdbc-ordered`** JDO metadata extension.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins”**

218].

- `meta-column`: An optional column that holds metadata about the collection, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of the owning class, not the table used to hold the collection elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the `jdbc-container-meta` JDO metadata extension.

### *Example 7.38. Using a Collection Mapping*

```
Java class:
public class Magazine
{
    private Set coverBlubs;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_COVERBLURBS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ELEMENT" type="varchar" size="255"/>
</table>

JDO metadata:
<class name="Magazine">
  <field name="coverBlubs">
    <collection element-type="String"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="coverBlubs">
    <jdbc-field-map type="collection" element-column="ELEMENT"
      table="MAG_COVERBLURBS" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions ...
  <field name="coverBlubs">
    <collection element-type="String"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="collection">
      <extension vendor-name="kodo" key="element-column" value="ELEMENT"/>
      <extension vendor-name="kodo" key="table" value="MAG_COVERBLURBS"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>
```

## 7.9.10. Many-to-Many Mapping

A many-to-many mapping represents a collection or array of related persistent objects using a join table. Each join table entry holds the primary key values of two related objects.

Many-to-many mappings can be one-sided or two-sided. In the latter case, the two fields involved in the relation both load from the join table. Only the "owning" side of the relation, as indicated by the **inverse-owner** JDO metadata extension, inserts and deletes join table entries, however.

The **mapping tool** creates many-to-many mappings by default on collections or arrays whose `element-type` is another persistent class, and who do not specify a `inverse-owner` or whose `inverse-owner` is another collection or array.

If your database defines a non-deferred foreign key constraint for the relation, you should be sure to set the **kodo.jdbc.ForeignKeyConstraints** property appropriately.

The many-to-many mapping has the following attributes:

- `type`: many-many
- `table`: The join table. This property is required.
- `element-column.<pk column>*`: Kodo JDO must be able to join this mapping's columns to the primary key columns of the related object. Each `element-column` attribute joins a column in the join table to the corresponding column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- `order-column`: The column that holds the position of each element within its collection. This property is optional. By default, the **mapping tool** orders fields as list or array types, and leaves collection and set fields unordered. You can control whether the mapping tool adds an order column with the **jdbc-ordered** JDO metadata extension. Only the owning side of a two-sided relation can be ordered.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- `meta-column`: An optional column that holds metadata about the collection, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the collection elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### *Example 7.39. Using a Many-to-Many Mapping*

```
Java class:
public class Magazine
{
    private Set articles;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>
```

```

<table name="ARTICLE">
  <column name="TITLE" type="varchar" size="127"/>
  <pk column="TITLE"/>
  ... columns for article fields ...
</table>

<table name="MAG_ARTS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ART_TITLE" type="varchar" size="127"/>
  <fk to-table="ARTICLE">
    <join column="ART_TITLE" to-column="TITLE"/>
  </fk>
</table>

JDO metadata:

<class name="Magazine">
  <field name="articles">
    <collection element-type="Article"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="articles">
    <jdbc-field-map type="many-many" element-column.TITLE="ART_TITLE"
      table="MAG_ARTS" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions ...
  <field name="articles">
    <collection element-type="Article"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="many-many">
      <extension vendor-name="kodo" key="element-column.TITLE"
        value="ART_TITLE"/>
      <extension vendor-name="kodo" key="table" value="MAG_ARTS"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

### ***Example 7.40. Using a Two-Sided Many-to-Many Mapping***

Notice that when we extend the relation to be two-sided in this example, the mappings for the `Magazine.articles` field do not change. The new `Article.magazines` field uses the same join table and simply inverts the `element-column` and `ref-column` in addition to signaling that the `articles` field is the relation's owner.

```

Java class:

public class Magazine
{
  private Set articles;
  ... class content ...
}

public class Article
{
  private Set magazines;
  ... class content ...
}

```

Schema:

```
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="ARTICLE">
  <column name="TITLE" type="varchar" size="127"/>
  <pk column="TITLE"/>
  ... columns for article fields ...
</table>

<table name="MAG_ARTS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ART_TITLE" type="varchar" size="127"/>
  <fk to-table="ARTICLE">
    <join column="ART_TITLE" to-column="TITLE"/>
  </fk>
</table>
```

JDO metadata:

```
<class name="Magazine">
  <field name="articles">
    <collection element-type="Article"/>
  </field>
  ... rest of field metadata ...
</class>

<class name="Article" objectid-class="ArticleId">
  <field name="title" primary-key="true"/>
  <field name="magazines">
    <collection element-type="Magazine"/>
    <extension vendor-name="kodo" key="inverse-owner" value="articles"/>
  </field>
  ... rest of field metadata ...
</class>
```

Mapping information using the mapping XML format:

```
<class name="Magazine">
  ... class mapping
  ... indicator mappings
  <field name="articles">
    <jdbc-field-map type="many-many" element-column.TITLE="ART_TITLE"
      table="MAG_ARTS" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

<class name="Article">
  ... class mapping
  ... indicator mappings
  <field name="magazines">
    <jdbc-field-map type="many-many" element-column.JDOID="MAG_ID"
      table="MAG_ARTS" ref-column.TITLE="ART_TITLE"/>
  </field>
  ... rest of field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Magazine">
  ... class extensions
  ... indicator extensions
  <field name="articles">
    <collection element-type="Article"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="many-many">
      <extension vendor-name="kodo" key="element-column.TITLE"
        value="ART_TITLE"/>
      <extension vendor-name="kodo" key="table" value="MAG_ARTS"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

<class name="Article">
  ... class extensions
  ... indicator extensions
  <field name="title" primary-key="true"/>
  <field name="magazines">
    <collection element-type="Magazine"/>
    <extension vendor-name="kodo" key="inverse-owner" value="articles"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="many-many">
```

```
<extension vendor-name="kodo" key="element-column.JDOID"
  value="MAG_ID"/>
<extension vendor-name="kodo" key="table" value="MAG_ARTS"/>
<extension vendor-name="kodo" key="ref-column.TITLE" value="ART_TITLE"/>
</extension>
</field>
... rest of field metadata ...
</class>
```

---

## 7.9.11. One-to-Many Mapping

A one-to-many mapping represents a field that holds a collection or array of related persistent objects, but does not use a join table. Instead, each related object record has a back-reference to its owning instance in the database. In a typical, two-sided one-to-many mapping, this database back-reference manifests itself in Java as a **one-to-one** relation back to the owning object. This inverse one-to-one is always the **inverse-owner** of the relation. In fact, the **mapping tool** installs a one-to-many mapping by default only when a collection or array field whose `element-type` is a persistent class declares a one-to-one field as its `inverse-owner`.

You can, however, have one-sided one-to-many mappings in which the back-reference only exists in the database, and is not represented by any Java field. The mapping tool never creates a one-sided one-to-many relation like this; you can only create it by writing the mapping data yourself, and then only on an existing schema that supports it.

If your database defines a non-deferred foreign key constraint for the relation, you should be sure to set the **kodo.jdbc.ForeignKeyConstraints** property appropriately.

The one-to-many mapping has the following attributes:

- `type`: one-many
- `table`: The table of the related class. This property is required.
- `order-column`: The column that holds the position of each element within its collection. This property is optional. By default, the **mapping tool** orders fields as list or array types, and leaves collection and set fields unordered. You can control whether the mapping tool adds an order column with the **jdbc-ordered** JDO metadata extension.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- `meta-column`: An optional column that holds metadata about the collection, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the collection elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.
- `class-criteria`: Whether or not to use the expected class of the related object as a criteria in the SQL issued to select its database record. Typically, this is not needed; Kodo can just use the foreign key value to select the desired record, and if your database data is valid, the record will correspond to an object of the correct class. Under some very rare mappings, however, you may need to select based on both foreign key and class -- for example, if you are using a relation that shares the inverse foreign key with another one-many with elements of a different subclass. In these rare cases, set this attribute to `true` to force Kodo to append class conditions to its SQL.

**Example 7.41. Using a One-to-Many Mapping**

```

Java class:

public class Magazine
{
    private Set articles;
    ... class content ...
}

public class Article
{
    private Magazine magazine;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
    <column name="JDOID" type="bigint"/>
    <pk column="JDOID"/>
    ... columns for magazine fields ...
</table>

<table name="ARTICLE">
    ... primary key columns ...
    <column name="MAG_ID" type="bigint"/>
    <fk to-table="MAGAZINE">
        <join column="MAG_ID" to-column="JDOID"/>
    </fk>
    ... columns for article fields ...
</table>

JDO metadata:

<class name="Magazine">
    <field name="articles">
        <collection element-type="Article"/>
        <extension vendor-name="kodo" key="inverse-owner" value="magazine"/>
    </field>
    ... rest of field metadata ...
</class>

<class name="Article">
    ... field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
    ... class mapping ...
    ... indicator mappings ...
    <field name="articles">
        <jdbc-field-map type="one-many" table="ARTICLE" ref-column.JDOID="MAG_ID"/>
    </field>
    ... rest of field mappings ...
</class>

<class name="Article">
    ... class mapping ...
    ... indicator mappings ...
    <field name="magazine">
        <jdbc-field-map type="one-one" column.JDOID="MAG_ID"/>
    </field>
    ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
    ... class extensions ...
    ... indicator extensions ...
    <field name="articles">
        <collection element-type="Article"/>
        <extension vendor-name="kodo" key="inverse-owner" value="magazine"/>
        <extension vendor-name="kodo" key="jdbc-field-map" value="one-many">
            <extension vendor-name="kodo" key="table" value="ARTICLE"/>
            <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>

<class name="Article">

```

```

... class extensions      ...
... indicator extensions  ...
<field name="magazine">
  <extension vendor-name="kodo" key="jdbc-field-map" value="one-one">
    <extension vendor-name="kodo" key="column.JDOID" value="MAG_ID"/>
  </extension>
</field>
... rest of field metadata ...
</class>

```

### Example 7.42. Using a One-Sided One-to-Many Mapping

The mappings for `Magazine.articles` are exactly the same in this one-sided relation example. The only difference is that the field no longer declares an `inverse-owner`, because the `Article.magazine` field does not exist in this example.

```

Java class:

public class Magazine
{
    private Set articles;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="ARTICLE">
  ... primary key columns      ...
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  ... columns for article fields ...
</table>

JDO metadata:

<class name="Magazine">
  <field name="articles">
    <collection element-type="Article"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings  ...
  <field name="articles">
    <jdbc-field-map type="one-many" table="ARTICLE" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions  ...
  <field name="articles">
    <collection element-type="Article"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="one-many">
      <extension vendor-name="kodo" key="table" value="ARTICLE"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.12. PC Collection Mapping

Fields that hold a collection or array of persistence-capable objects of an unknown type use the PC collection mapping. It is the default mapping when the `element-type` of a collection or array is a user-defined interface, is a horizontally mapped class (Section 7.6.4, “Horizontal Inheritance Mapping” [224]), or when the `element-type` JDO metadata extension is set to `PersistenceCapable`. The PC collection mapping's schema structure is similar to that of the **collection mapping**, but the element column stores the stringified value of each related persistent object's oid value. You cannot query across this mapping.

The PC collection mapping has the following attributes:

- `type`: `pc-collection`
- `table`: The table where the stringified oid values of the collection elements are stored. This property is required.
- `element-column`: The column that holds each stringified oid. This property is required.
- `order-column`: The column that holds the position of each element within its collection. This property is optional. By default, the **mapping tool** orders fields as list or array types, and leaves collection and set fields unordered. You can control whether the mapping tool adds an order column with the `jdbc-ordered` JDO metadata extension.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see Section 7.5.2, “Non-Standard Joins” [218].
- `meta-column`: An optional column that holds metadata about the collection, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the collection elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the `jdbc-container-meta` JDO metadata extension.

### Example 7.43. Using a PC Collection Mapping

```
Java class:
public class Magazine
{
    private Set items;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_ITEMS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ELEMENT" type="varchar" size="255"/>
</table>

JDO metadata:
<class name="Magazine">
  <field name="items">
    <collection element-type="IMagazineItem"/>
  </field>
```

```
... rest of field metadata ...
</class>
```

Mapping information using the mapping XML format:

```
<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="items">
    <jdbc-field-map type="pc-collection" element-column="ELEMENT"
      table="MAG_ITEMS" ref-column.JDOID="MAG_ID" />
  </field>
  ... rest of field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Magazine">
  ... class extensions    ...
  ... indicator extensions ...
  <field name="items">
    <collection element-type="IMagazineItem" />
    <extension vendor-name="kodo" key="jdbc-field-map" value="pc-collection">
      <extension vendor-name="kodo" key="element-column" value="ELEMENT" />
      <extension vendor-name="kodo" key="table" value="MAG_ITEMS" />
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID" />
    </extension>
  </field>
  ... rest of field metadata ...
</class>
```

## 7.9.13. Map Mapping

The map mapping represents a map in which the keys and values are simple types (primitive wrappers, dates, or strings). It has the following attributes:

- **type**: map
- **table**: The table where the map entries are stored. This property is required.
- **key-column**: The column that holds each map entry key. This property is required.
- **value-column**: The column that holds each map entry value. This property is required.
- **ref-column.<pk column>\***: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each **ref-column** attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- **ref-constant.<column>\***: Similar to the **ref-column** attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- **meta-column**: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.44. Using a Map Mapping

Java class:

```

public class Magazine
{
    private Map index;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_INDEX">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="TOPIC" type="varchar" size="255"/>
  <column name="PAGENUMBER" type="integer"/>
</table>

JDO metadata:

<class name="Magazine">
  <field name="index">
    <map key-type="String" value-type="Integer"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping
  ... indicator mappings
  <field name="index">
    <jdbc-field-map type="map" key-column="TOPIC" value-column="PAGENUMBER"
      table="MAG_INDEX" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions
  ... indicator extensions
  <field name="index">
    <map key-type="String" value-type="Integer"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="map">
      <extension vendor-name="kodo" key="key-column" value="TOPIC"/>
      <extension vendor-name="kodo" key="value-column" value="PAGENUMBER"/>
      <extension vendor-name="kodo" key="table" value="MAG_INDEX"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.14. N-to-Many Map Mapping

A map field in which the map keys are a simple type (primitive wrapper, string) and the map values are related persistent objects. This mapping has the following attributes:

- `type`: n-many-map
- `table`: The table that holds the map entries. This property is required.
- `key-column`: The column that holds each map entry key. This property is required.
- `value-column.<pk column>*`: Kodo JDO must be able to join this mapping's map value columns to the columns of

the related object. Each `value-column` attribute joins a value column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.

- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.45. Using an N-to-Many Map Mapping

```
Java class:
public class Magazine
{
    private Map featuredArticles;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="ARTICLE">
  <column name="TITLE" type="varchar" size="127"/>
  <pk column="TITLE"/>
  ... columns for article fields ...
</table>

<table name="FEATUREDARTS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="BLURB" type="varchar" size="255"/>
  <column name="ART_TITLE" type="varchar" size="127"/>
  <fk to-table="ARTICLE">
    <join column="ART_TITLE" to-column="TITLE"/>
  </fk>
</table>

JDO metadata:
<class name="Magazine">
  <field name="featuredArticles">
    <map key-type="String" value-type="Article"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping
  ... indicator mappings
  <field name="featuredArticles">
    <jdbc-field-map type="n-many-map" key-column="BLURB"
      value-column.TITLE="ART_TITLE" table="FEATUREDARTS"
      ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>
```

Mapping information using JDO metadata extensions:

```
<class name="Magazine">
... class extensions
... indicator extensions
<field name="featuredArticles">
  <map key-type="String" value-type="Article"/>
  <extension vendor-name="kodo" key="jdbc-field-map" value="n-many-map">
    <extension vendor-name="kodo" key="key-column" value="BLURB"/>
    <extension vendor-name="kodo" key="value-column.TITLE"
      value="ART_TITLE"/>
    <extension vendor-name="kodo" key="table" value="FEATUREDARTS"/>
    <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
  </extension>
</field>
... rest of field metadata ...
</class>
```

## 7.9.15. Many-to-N Map Mapping

A map field in which the map values are a simple type (primitive wrapper, string) and the map keys are related persistent objects. This mapping has the following attributes:

- `type`: many-n-map
- `table`: The table that holds the map entries. This property is required.
- `key-column.<pk column>*`: Kodo JDO must be able to join this mapping's map key columns to the columns of the related object. Each `key-column` attribute joins a key column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- `value-column`: The column that holds each map entry value. This property is required.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.46. Using a Many-to-N Map Mapping

Java class:

```
public class Magazine
{
  private Map images;
  ... class content ...
}
```

Schema:

```
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
```

```

... columns for magazine fields ...
</table>

<table name="IMAGE">
  <column name="ID" type="bigint"/>
  <pk column="ID"/>
  ... columns for image fields ...
</table>

<table name="MAG_IMAGES">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="IMAGE_ID" type="bigint"/>
  <fk to-table="IMAGE">
    <join column="IMAGE_ID" to-column="ID"/>
  </fk>
  <column name="CAPTION" type="varchar" size="255"/>
</table>

JDO metadata:

<class name="Magazine">
  <field name="images">
    <map key-type="Image" value-type="String"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="featuredArticles">
    <jdbc-field-map type="many-n-map"
      key-column.ID="IMAGE_ID" value-column="CAPTION"
      table="MAG_IMAGES" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions ...
  <field name="images">
    <map key-type="Image" value-type="String"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="many-n-map">
      <extension vendor-name="kodo" key="key-column.ID" value="IMAGE_ID"/>
      <extension vendor-name="kodo" key="value-column" value="CAPTION"/>
      <extension vendor-name="kodo" key="table" value="MAG_IMAGES"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.16. Many-to-Many Map Mapping

A map in which both the keys and values are relations to other persistent objects use the many-to-many map mapping. This mapping has the following attributes:

- `type`: many-many-map
- `table`: The table that holds the map entries. This property is required.
- `key-column.<pk column>*`: Kodo JDO must be able to join this mapping's map key columns to the columns of the related object. Each `key-column` attribute joins a key column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.

- `value-column.<pk column>*`: Kodo JDO must be able to join this mapping's map value columns to the columns of the related object. Each `value-column` attribute joins a value column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, "Non-Standard Joins" [218]**.
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the `jdbc-container-meta` JDO metadata extension.

### Example 7.47. Using a Many-to-Many Map Mapping

```

Java class:
public class Magazine
{
    private Map articleImages;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="ARTICLE">
  <column name="TITLE" type="varchar" size="127"/>
  <pk column="TITLE"/>
  ... columns for article fields ...
</table>

<table name="IMAGE">
  <column name="ID" type="bigint"/>
  <pk column="ID"/>
  ... columns for image fields ...
</table>

<table name="ARTICLEIMAGES">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ART_TITLE" type="varchar" size="127"/>
  <fk to-table="ARTICLE">
    <join column="ART_TITLE" to-column="TITLE"/>
  </fk>
  <column name="IMAGE_ID" type="bigint"/>
  <fk to-table="IMAGE">
    <join column="IMAGE_ID" to-column="ID"/>
  </fk>
</table>

JDO metadata:
<class name="Magazine">
  <field name="articleImages">
    <map key-type="Article" value-type="Image"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">

```

```

... class mapping
... indicator mappings
<field name="articleImages">
  <jdbc-field-map type="many-many-map"
    key-column.TITLE="ART_TITLE" value-column.ID="IMAGE_ID"
    table="ARTICLEIMAGES" ref-column.JDOID="MAG_ID"/>
</field>
... rest of field mappings ...
</class>

```

Mapping information using JDO metadata extensions:

```

<class name="Magazine">
... class extensions
... indicator extensions
<field name="articleImages">
  <map key-type="Article" value-type="Image"/>
  <extension vendor-name="kodo" key="jdbc-field-map" value="many-many-map">
    <extension vendor-name="kodo" key="key-column.TITLE"
      value="ART_TITLE"/>
    <extension vendor-name="kodo" key="value-column.ID"
      value="IMAGE_ID"/>
    <extension vendor-name="kodo" key="table" value="ARTICLEIMAGES"/>
    <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
  </extension>
</field>
... rest of field metadata ...
</class>

```

## 7.9.17. PC Map Mapping

A PC map is a map in which both the key and value types are unknown persistence-capable classes. This is the default mapping for maps fields whose `key-type` and `value-type` are user-defined interfaces, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), or who have set the **key-type**, **value-type** JDO metadata extensions to `PersistenceCapable`. The key column and value column of this mapping hold the stringified oid value of each map entry's key and value object, respectively.

This mapping has the following attributes:

- `type`: `pc-map`
- `table`: The table where the map entries are stored. This property is required.
- `key-column`: The column that holds the stringified oid of each map entry key. This property is required.
- `value-column`: The column that holds the stringified oid of each map entry value. This property is required.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see [Section 7.5.2, “Non-Standard Joins” \[218\]](#).
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.48. Using a PC Map Mapping

```

Java class:

public class Magazine
{
    private Map items;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_ITEMS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="ITEMTYPE" type="varchar" size="255"/>
  <column name="ITEM" type="varchar" size="255"/>
</table>

JDO metadata:

<class name="Magazine">
  <field name="items">
    <map key-type="IMagazineItemType" value-type="IMagazineItem"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping      ...
  ... indicator mappings ...
  <field name="items">
    <jdbc-field-map type="pc-map" key-column="ITEMTYPE"
      value-column="ITEM" table="MAG_ITEMS" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions      ...
  ... indicator extensions ...
  <field name="items">
    <map key-type="IMagazineItemType" value-type="IMagazineItem"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="pc-map">
      <extension vendor-name="kodo" key="key-column" value="ITEMTYPE"/>
      <extension vendor-name="kodo" key="value-column" value="ITEM"/>
      <extension vendor-name="kodo" key="table" value="MAG_ITEMS"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.18. N-to-PC Map Mapping

The n-to-PC map mapping represents a map whose keys are a simple type (primitive wrapper, date, string) and whose values are an unknown persistence-capable type. This is the default mapping for maps with simple keys and user-defined interface values, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), or fields who have set the **value-type** JDO metadata extension to `PersistenceCapable`. The value column of this mapping holds the stringified oid value of each map entry's value object.

This mapping has the following attributes:

- `type: n-pc-map`
- `table`: The table where the map entries are stored. This property is required.
- `key-column`: The column that holds the each map entry key. This property is required.
- `value-column`: The column that holds the stringified oid of each map entry value. This property is required.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the `jdbc-container-meta` JDO metadata extension.

### *Example 7.49. Using an N-to-PC Map Mapping*

```

Java class:
public class Magazine
{
    private Map figures;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="MAG_FIGURES">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="PAGENUMBER" type="integer"/>
  <column name="FIGURE" type="varchar" size="255"/>
</table>

JDO metadata:
<class name="Magazine">
  <field name="figures">
    <map key-type="Integer" value-type="IFigure"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping
  ... indicator mappings
  <field name="figures">
    <jdbc-field-map type="n-pc-map" key-column="PAGENUMBER"
      value-column="FIGURE" table="MAG_FIGURES" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

```

```

<class name="Magazine">
  ... class extensions
  ... indicator extensions
  <field name="figures">
    <map key-type="Integer" value-type="IFigure"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="n-pc-map">
      <extension vendor-name="kodo" key="key-column" value="PAGENUMBER"/>
      <extension vendor-name="kodo" key="value-column" value="FIGURE"/>
      <extension vendor-name="kodo" key="table" value="MAG_FIGURES"/>
      <extension vendor-name="kodo" key="ref-column.JDROID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.19. PC-to-N Map Mapping

The PC-to-n map mapping represents a map whose keys are an unknown persistence-capable type and whose values are a simple type (primitive wrapper, date, string). This is the default mapping for maps with simple values and user-defined interface keys, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), or fields who have set the **key-type** JDO metadata extension to `PersistenceCapable`. The key column of this mapping holds the stringified oid value of each map entry's key object.

This mapping has the following attributes:

- `type: pc-n-map`
- `table`: The table where the map entries are stored. This property is required.
- `key-column`: The column that holds the stringified oid of each map entry key. This property is required.
- `value-column`: The column that holds each map entry value. This property is required.
- `ref-column.<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each `ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see [Section 7.5.2, “Non-Standard Joins” \[218\]](#).
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.50. Using a PC-to-N Map Mapping

```

Java class:
public class Magazine
{
    private Map figures;
    ... class content ...
}

Schema:
<table name="MAGAZINE">

```

```

    <column name="JDOID" type="bigint"/>
    <pk column="JDOID"/>
    ... columns for magazine fields ...
</table>

<table name="MAG_FIGURES">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="FIGURE" type="varchar" size="255"/>
  <column name="CAPTION" type="varchar" size="255"/>
</table>

JDO metadata:

<class name="Magazine">
  <field name="figures">
    <map key-type="IFigure" value-type="String"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping ...
  ... indicator mappings ...
  <field name="figures">
    <jdbc-field-map type="pc-n-map" key-column="FIGURE"
      value-column="CAPTION" table="MAG_FIGURES" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions ...
  ... indicator extensions ...
  <field name="figures">
    <map key-type="IFigure" value-type="String"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="pc-n-map">
      <extension vendor-name="kodo" key="key-column" value="FIGURE"/>
      <extension vendor-name="kodo" key="value-column" value="CAPTION"/>
      <extension vendor-name="kodo" key="table" value="MAG_FIGURES"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.20. PC-to-Many Map Mapping

A map field in which the map keys are an unknown persistence-capable type and the map values are related persistent objects. Unknown persistence-capable types include user-defined interfaces, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), and any field whose **key-type** JDO metadata extension is set to `PersistenceCapable`. This mapping has the following attributes:

- **type**: `pc-many-map`
- **table**: The table that holds the map entries. This property is required.
- **key-column**: The column that holds the stringified oid of each map entry key. This property is required.
- **value-column**. `<pk column>*`: Kodo JDO must be able to join this mapping's map value columns to the columns of the related object. Each **value-column** attribute joins a value column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- **ref-column**. `<pk column>*`: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each

`ref-column` attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.

- `ref-constant.<column>*`: Similar to the `ref-column` attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see **Section 7.5.2, “Non-Standard Joins” [218]**.
- `meta-column`: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the `jdbc-container-meta` JDO metadata extension.

### Example 7.51. Using a PC-to-Many Map Mapping

```
Java class:
public class Magazine
{
    private Map chapterMarkers;
    ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

<table name="CHAPTER">
  <column name="TITLE" type="varchar" size="127"/>
  <pk column="TITLE"/>
  ... columns for chapter fields ...
</table>

<table name="CHAPTERMARKERS">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="MARKER" type="varchar" size="255"/>
  <column name="CHAPT_TITLE" type="varchar" size="127"/>
  <fk to-table="CHAPTER">
    <join column="CHAPT_TITLE" to-column="TITLE"/>
  </fk>
</table>

JDO metadata:
<class name="Magazine">
  <field name="chapterMarkers">
    <map key-type="IMarker" value-type="Chapter"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:
<class name="Magazine">
  ... class mapping ...
  ... indicator mappings ...
  <field name="chapterMarkers">
    <jdbc-field-map type="pc-many-map" key-column="MARKER"
      value-column="CHAPT_TITLE" table="CHAPTERMARKERS"
      ref-column="JDOID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:
<class name="Magazine">
  ... class extensions ...
  ... indicator extensions ...
  <field name="chapterMarkers">
```

```

<map key-type="IMarker" value-type="Chapter"/>
<extension vendor-name="kodo" key="jdbc-field-map" value="pc-many-map">
  <extension vendor-name="kodo" key="key-column" value="MARKER"/>
  <extension vendor-name="kodo" key="value-column.TITLE"
    value="CHAPT_TITLE"/>
  <extension vendor-name="kodo" key="table" value="CHAPTERMARKERS"/>
  <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
</extension>
</field>
... rest of field metadata ...
</class>

```

## 7.9.21. Many-to-PC Map Mapping

A map field in which the map keys are related persistent objects and the map values are an unknown persistence-capable type. Unknown persistence-capable types include user-defined interfaces, horizontally mapped classes ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#)), and any field whose **value-type** JDO metadata extension is set to `PersistenceCapable`. This mapping has the following attributes:

- **type**: `many-pc-map`
- **table**: The table that holds the map entries. This property is required.
- **key-column.<pk column>\***: Kodo JDO must be able to join this mapping's map key columns to the columns of the related object. Each **key-column** attribute joins a key column in the map table to the corresponding primary key column in the table of the related object's class. See the previous discussion of **join attributes** for details on joins.
- **value-column**: The column that holds the stringified oid of each map entry value. This property is required.
- **ref-column.<pk column>\***: Kodo JDO must be able to join this mapping's table to the table of the owning class. Each **ref-column** attribute joins a column in this mapping's table to the corresponding column in the owning class' table. See the previous discussion of **join attributes** for details on joins.
- **ref-constant.<column>\***: Similar to the **ref-column** attribute, but used when the join relies on a column in the joined-to table having a constant value. For more information on constant joins, see [Section 7.5.2, “Non-Standard Joins” \[218\]](#).
- **meta-column**: An optional column that holds metadata about the map, such as whether it is null or simply empty. If present, this column must be able to hold numeric data and must be in the declared table of of the owning class, not the table used to hold the map elements. The mapping tool never adds a metadata column by default, but you can indicate that you would like one with the **jdbc-container-meta** JDO metadata extension.

### Example 7.52. Using a Many-to-PC Map Mapping

```

Java class:
public class Magazine
{
  private Map imageCategories;
  ... class content ...
}

Schema:
<table name="MAGAZINE">
  <column name="JDOID" type="bigint"/>
  <pk column="JDOID"/>
  ... columns for magazine fields ...
</table>

```

```

<table name="IMAGE">
  <column name="ID" type="bigint"/>
  <pk column="ID"/>
  ... columns for image fields ...
</table>

<table name="MAG_IMAGES">
  <column name="MAG_ID" type="bigint"/>
  <fk to-table="MAGAZINE">
    <join column="MAG_ID" to-column="JDOID"/>
  </fk>
  <column name="IMAGE_ID" type="bigint"/>
  <fk to-table="IMAGE">
    <join column="IMAGE_ID" to-column="ID"/>
  </fk>
  <column name="CATEGORY" type="varchar" size="255"/>
</table>

JDO metadata:

<class name="Magazine">
  <field name="images">
    <map key-type="Image" value-type="ICategory"/>
  </field>
  ... rest of field metadata ...
</class>

Mapping information using the mapping XML format:

<class name="Magazine">
  ... class mapping ...
  ... indicator mappings ...
  <field name="imageCategories">
    <jdbc-field-map type="many-pc-map"
      key-column.ID="IMAGE_ID" value-column="CATEGORY"
      table="MAG_IMAGES" ref-column.JDOID="MAG_ID"/>
  </field>
  ... rest of field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Magazine">
  ... class extensions ...
  ... indicator extensions ...
  <field name="imageCategories">
    <map key-type="Image" value-type="ICategory"/>
    <extension vendor-name="kodo" key="jdbc-field-map" value="many-pc-map">
      <extension vendor-name="kodo" key="key-column.ID" value="IMAGE_ID"/>
      <extension vendor-name="kodo" key="value-column" value="CATEGORY"/>
      <extension vendor-name="kodo" key="table" value="MAG_IMAGES"/>
      <extension vendor-name="kodo" key="ref-column.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

## 7.9.22. Custom Field Mapping

Kodo JDO allows you to create your own field mappings. A custom field mapping can override any or all of the CRUD operations for the field: Create, Retrieve, Update, Delete. All mappings must extend, directly or indirectly, from `kodo.jdbc.meta.FieldMapping`.

The `jdbc-field-map-name` JDO metadata extension tells the **mapping tool** which field mapping to install. If you write mappings by hand rather than with the mapping tool, simply specify the full class name of your custom mapping in the `type` attribute of the mapping XML.

The `samples/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.

## 7.9.23. Externalization

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 mappings** by serializing the field value. 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. Often, however, a custom mapping is overkill. Many field types that are not supported by JDO can be easily represented as a type that is directly supported; thus, Kodo provides the externalization service. Externalization allows you to specify methods that will externalize a field value to a supported type on store and then rebuild the value from its externalized form on load.

The field-level **externalizer** metadata extension contains the name of a method that will be invoked to convert the field into its external form 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. Each method can also take an optional `PersistenceManager` parameter. The return value of the method is the field's external form. By default, Kodo assumes that all named methods belong to the field value's class (or its superclasses). You can, however, specify static methods of other classes using the format `<class-name>.<method-name>`.

Given a field of type `CustomType` that externalizes to a string, the table below demonstrates several possible externalizer methods and their corresponding metadata extensions.

**Table 7.1. Externalizer Options**

Method	Extension
<code>public String CustomType.toString()</code>	<code>&lt;extension vendor-name="kodo" key="externalizer" value="toString"/&gt;</code>
<code>public String CustomType.toString(PersistenceManager pm)</code>	<code>&lt;extension vendor-name="kodo" key="externalizer" value="toString"/&gt;</code>
<code>public static String AnyClass.toString(CustomType ct)</code>	<code>&lt;extension vendor-name="kodo" key="externalizer" value="AnyClass.toString"/&gt;</code>
<code>public static String AnyClass.toString(CustomType ct, PersistenceManager pm)</code>	<code>&lt;extension vendor-name="kodo" key="externalizer" value="AnyClass.toString"/&gt;</code>

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

Given a field of type `CustomType` that externalizes to a string, the table below demonstrates several possible factory methods and their corresponding metadata extensions.

**Table 7.2. Factory Options**

Method	Extension
<code>public CustomType(String str)</code>	none
<code>public static CustomType CustomType.fromString(String str)</code>	<code>&lt;extension vendor-name="kodo" key="factory" value="fromString"/&gt;</code>
<code>public static CustomType CustomType.fromString(String str, PersistenceManager pm)</code>	<code>&lt;extension vendor-name="kodo" key="factory" value="fromString"/&gt;</code>
<code>public static CustomType AnyClass.fromString(String str)</code>	<code>&lt;extension vendor-name="kodo" key="factory" value="AnyClass.fromString"/&gt;</code>
<code>public static CustomType AnyClass.fromString(String str, PersistenceManager pm)</code>	<code>&lt;extension vendor-name="kodo" key="factory" value="AnyClass.fromString"/&gt;</code>

Note that by default, fields whose type is not supported by JDO are neither persistent nor in the default fetch group, so you will have to specify the `persistence-modifier` and `default-fetch-group` metadata attributes explicitly.

### Note

As with fields that are stored through serialization, it is not possible to detect changes to the internal state of the field object. If your field is not a standard JDO type and you change the field object without assigning a new value to the field, you will have to mark the field dirty yourself using `JDOHelper.makeDirty`, unless you have created a custom proxy for the field. See the `samples/proxies` directory for examples of custom proxies.

You can externalize a field to virtually any value that is supported by Kodo's field mappings (the **embedded one-to-one** mapping is the single exception; you must declare your field to be a persistence-capable type in order to embed it). This means that a field can externalize to something as simple as a primitive, something as complex as a collection or map of persistence-capable objects, or anything in between. If you do choose to externalize to a collection or map, the **element-type**, **key-type**, and **value-type** metadata extensions allow you to specify the corresponding metadata for the externalized form of your field. If the external form of your field is a persistence-capable object, or contains persistence-capable objects, Kodo will correctly include the objects in its persistence-by-reachability algorithms and its delete-dependent algorithms (meaning you can take advantage of the **dependent** family of metadata extensions).

The example below demonstrates a few forms of externalization. See the `samples/externalization` directory of your Kodo distribution for a working example of externalizing many different field types.

### Example 7.53. Using Externalization

```
public class Magazine
{
    private Class    cls;
    private URL      url;
    private CustomType customType;

    public static Map mapFromCustomType (CustomType customType)
    {
        ... logic to pack custom type into a map ...
    }

    public static CustomType mapToCustomType (Map map)
    {
        ... logic to create custom type from a map ...
    }

    ... class content ...
}

<class name="Magazine">
  <field name="cls" persistence-modifier="persistent"
    default-fetch-group="true">
```

```

<!-- use Class.getName and Class.forName to go to/from strings -->
<extension vendor-name="kodo" key="externalizer" value="getName"/>
<extension vendor-name="kodo" key="factory" value="forName"/>
</field>
<field name="url" persistence-modifier="persistent"
  default-fetch-group="true">
  <!-- use URL.getExternalForm for externalization; no -->
  <!-- factory; we can rely on the URL string constructor -->
  <extension vendor-name="kodo" key="externalizer"
    value="toExternalForm"/>
</field>
<field name="customType" persistence-modifier="persistent">
  <!-- use our custom methods; notice how we use the -->
  <!-- key-type and value-type extensions to specify -->
  <!-- the metadata for our externalized map -->
  <extension vendor-name="kodo" key="externalizer"
    value="Magazine.mapFromCustomType"/>
  <extension vendor-name="kodo" key="factory"
    value="Magazine.mapToCustomType"/>
  <extension vendor-name="kodo" key="key-type" value="String"/>
  <extension vendor-name="kodo" key="value-type" value="String"/>
</field>
... field metadata ...
</class>

```

You can also query externalized fields using parameters. Pass in a value of the field type when executing the query. Kodo will externalize the parameter using the externalize method named in your metadata, and compare the externalized parameter with the externalized value stored in the database. As a shortcut, Kodo also allows you to use parameters or literals of the field's externalized type in queries, as demonstrated in the example below.

### Note

Currently, queries are limited to fields that externalize to a primitive, primitive wrapper, string, or date, due to constraints on query syntax.

### Example 7.54. Querying Externalization Fields

Assume the Magazine class has the same fields as in the previous example.

```

// you can query using parameters
Query q = pm.newQuery (Magazine.class, "url == :u");
Collection results = (Collection)
  q.execute (new URL ("http://www.solarmetric.com"));

// or as a shortcut, you can use the externalized form directly
q = pm.newQuery (Magazine.class, "url == 'http://www.solarmetric.com'");
results = (Collection) q.execute ();

```

## 7.9.24. External Values

Externalization often takes simple constant values and transforms them to constant values of a different type. An example would be storing a boolean field as a char, where `true` and `false` would be stored in the database as 'T' and 'F' respectively.

Kodo allows the ability to define these constant values in metadata, so that the field will behave like **field externalization** without requiring a custom externalizer/factory. External values supports translation of pre-defined simple types (primitives, primitive wrappers, Dates, and Strings), to other pre-defined simple typed values.

All constant values must be defined in the `external-values` field extension. The values are defined in a similar format as

**configuration plugins**, except that the value pairs are Java and datastore values respectively. Extending the previous example, one would use the extension value: `true=T`, `false=F`. If the datatype of the datastore value is different from the field's, the datastore type can be defined using the **type** field extension.

### *Example 7.55. Using External Values*

This example uses the `external-values` and `type` field extension to transform a `String` field to an integer at the database.

```
public class Magazine
{
    private String sizeWidth;
    .... class content ...
}

Schema:

<table name="MAGAZINE">
    ... primary key columns
    <column name="SIZE_WIDTH" type="integer"/>
    ... columns for magazine fields ...
</table>

JDO metadata:

<class name="Magazine">
    <field name="sizeWidth">
        <extension vendor-name="kodo" key="type" value="int"/>
        <extension vendor-name="kodo" key="external-values"
            value="SMALL=5,MEDIUM=8,LARGE=10"/>
    </field>
    ... field metadata ...
</class>
```

---

# Chapter 8. Schema Information

Kodo JDO stores persistent objects in relational database tables. This raises two important questions:

1. How does Kodo JDO get information about your database schema?
2. How do your persistent class definitions and your database schema stay synchronized?

This chapter explores the answers to these questions.

---

## 8.1. Schema Reflection

Kodo JDO needs information about your database schema for two reasons. First, it needs schema information at runtime to validate that your schema is compatible with your persistent class definitions, and to perform advanced actions like foreign key constraint analysis. Second, Kodo JDO requires schema information during development so that it can manipulate the schema to match your object model. Kodo JDO uses the `SchemaFactory` interface to provide runtime mapping information, and the schema generator tool for development-time data. Each is presented below.

---

### 8.1.1. Schemas List

By default, the tools and processes covered in this chapter act on all the schemas your JDBC driver can "see". You can limit the schemas and tables Kodo acts on with the `kodo.jdbc.Schemas` configuration property. This property accepts a comma-separated list of schemas and tables. To list a schema, list its name. To list a table, list its full name in the form `<schema-name>.<table-name>`. If a table does not have a schema or you do not know its schema, list its name as `.<table-name>` (notice the preceding '.'). For example, to list the `BUSOBSJS` schema, the `ADDRESS` table in the `GENERAL` schema, and the `SYS-TEM_INFO` table, regardless of what schema it is in, use the string:

```
kodo.jdbc.Schemas: BUSOBSJS,GENERAL.ADDRESS, .SYSTEM_INFO
```

When Kodo creates new tables, it will place them in the first schema listed.

#### Note

Some databases are case-sensitive with respect to schema and table names. Oracle, for example, requires names in all upper case.

---

### 8.1.2. Schema Factory

Kodo JDO relies on the `kodo.jdbc.SchemaFactory` interface for runtime schema information. You can control the schema factory Kodo JDO uses through the `kodo.jdbc.SchemaFactory` property. There are several built-in options to choose from:

- `native`: This is the default schema factory, and the one recommended for most users. It is an alias for the `kodo.jdbc.schema.LazySchemaFactory`. As persistent classes are loaded by the application, Kodo JDO reads their JDO metadata and object-relational mapping information. This factory uses the `java.sql.DatabaseMetaData` interface to reflect on the schema and ensure that it is consistent with the mapping data being read. Because the factory doesn't reflect on a table definition until that table is mentioned by the mapping information, we call its class "lazy".

- **dynamic**: This is an alias for the `kodo.jdbc.schema.DynamicSchemaFactory`. The `DynamicSchemaFactory` is the most performant schema factory, because it does not validate mapping information against the existing schema. Instead, it always assumes all object-relational mapping information is correct, and dynamically builds up an in-memory representation of the schema from the mapping data. You may want to use this factory if your JDBC driver doesn't support the `java.sql.DatabaseMetaData` interface, or you trust that your mapping data and schema are always in synch. You may also want to use this schema factory to prevent validation if you use database views for certain classes. Most JDBC drivers do not report the structure of views, and so any classes mapped to views will probably fail validation.

### Note

The dynamic schema factory cannot auto-detect foreign keys, column defaults, non-nullable columns, and other database information. This means that if the schema has any foreign keys that are not deferred, Kodo will not know to re-order SQL statements to satisfy the foreign keys constraints and some inserts will fail; in these cases, the `file` or `native` schema factory will need to be used instead.

- **db**: This is an alias for the `kodo.jdbc.schema.DBSchemaFactory`. This schema factory stores schema information as an XML document in a database table it creates for this purpose. If your JDBC driver doesn't support the `java.sql.DatabaseMetaData` standard interface, but you still want some schema validation to occur at runtime, you might use this factory. It is not recommended for most users, though, because it is easy for the stored XML schema definition to get out-of-synch with the actual database. This factory accepts the following properties:
  - **TableName**: The name of the table to create to store schema information. Defaults to `JDO_SCHEMA`.
- **file**: This is an alias for the `kodo.jdbc.schema.FileSchemaFactory`. This factory is a lot like the `DBSchemaFactory`, and has the same advantages and disadvantages. Instead of storing its XML schema definition in a database table, though, it stores it in a file. This factory accepts the following properties:
  - **FileName**: The resource name of the XML schema file. By default, the factory looks for a resource called `package.schema`, located in any top-level directory of the `CLASSPATH` or in the top level of any jar in your `CLASSPATH`.

You can switch freely between schema factories at any time. The XML file format used by some factories is detailed in [Section 8.3, “XML Schema Format”](#) [282]. Some of the factories are configurable; see their **Javadoc** for details. Finally, as with any Kodo JDO plugin, you can implement your own schema factory if you have needs not met by the existing options.

## 8.1.3. Schema Generator

While the schema factory provides schema information to runtime components, most development tools rely on the schema generator behind-the-scenes. The schema generator uses the JDBC driver's `java.sql.DatabaseMetaData` implementation to build up an internal representation of your database schema. This internal representation can then be dumped to an XML file, which in turn can be manipulated by other tools, most notably the **schema tool**. The XML format used by the schema generator is detailed [below](#). Some schema factories also invoke the schema generator programmatically and directly use its Java object view of schema components.

As a user, you will rarely if ever use the schema generator directly. Should the need arise, however, you can invoke the schema generator via the `schemagen` shell/bat script included in the Kodo JDO distribution, or via its Java class, `kodo.jdbc.schema.SchemaGenerator`.

### Example 8.1. Using the Schema Generator

```
schemagen -f schema.xml
```

The schema generator accepts the following flags (in addition to the universal flags discussed in the **Configuration Framework** chapter):

- `-indexes/-ix <true/t | false/f>`: Whether to generate information about table indexes. Defaults to `true`.
- `-primaryKeys/-pk <true/t | false/f>` : Whether to generate information about primary keys. Defaults to `true`.
- `-foreignKeys/-fk <true/t | false/f>` : Whether to generate information about foreign keys. Defaults to `true`.
- `-kodoTables/-kt <true/t | false/f>`: Whether to generate information about special Kodo-generated tables such as sequence tables. Defaults to `false`.
- `-schemas/-s <schema and table names>` : A comma-separated list of schema and table names to generate information on. Each element of the list must follow the naming conventions for the `kodo.jdbc.Schemas` property. In fact, if this flag is omitted, it defaults to the value of the `Schemas` property. If the `Schemas` property is not defined, information for all schemas will be generated.
- `-file/-f <stdout | output file>` : The name of the output file to write the XML schema definition to. If the file names a resource in the `CLASSPATH`, data will be written to that resource. Use `stdout` to write the XML to standard output. Defaults to `stdout`.

## 8.2. Schema Tool

---

Most users will only use the schema tool indirectly, through the interface provided by the **mapping tool**. You may find, however, that the schema tool is a powerful utility in its own right.

The schema tool's function is to take in an XML schema definition, calculate the differences between the XML and the existing database schema, and apply the necessary changes to make the database match the XML. The **XML format** used by the schema tool abstracts away the differences between SQL dialects used by different database vendors. The tool also automatically adapts its SQL to meet foreign key dependencies. Thus the schema tool is very useful as a general way to manipulate schemas.

You can invoke the schema tool through the `schematool` shell/bat script included in the Kodo JDO distribution, or through its Java class, `kodo.jdbc.schema.SchemaTool`. In addition to the universal flags of the **configuration framework**, the schema tool accepts the following command line arguments:

- `-ignoreErrors/-i <true/t | false/f>` : If `false`, an exception will be thrown if the tool encounters any database errors. Defaults to `false`.
- `-file/-f <stdout | output file>`: Use this option to write a SQL script for the planned schema modifications, rather than committing them to the database. When used in conjunction with the `export` action, the named file will be used to write the exported schema XML. If the file names a resource in the `CLASSPATH`, data will be written to that resource. Use `stdout` to write to standard output. Defaults to `stdout`.
- `-dropTables/-dt <true/t | false/f>`: Set this option to `true` to drop tables that appear to be unused during `retain` and `refresh` actions. Defaults to `true`.
- `-kodoTables/-kt <true/t | false/f>`: Whether to generate information about special Kodo-generated tables such as sequence tables as part of the database schema. Defaults to `false`.
- `-indexes/-ix <true/t | false/f>`: Whether to manipulate indexes on existing tables. Defaults to `true`.
- `-primaryKeys/-pk <true/t | false/f>` : Whether to manipulate primary keys on existing tables. Defaults to `true`.

- `-foreignKeys/-fk <true/t | false/f>` : Whether to manipulate foreign keys on existing tables. Defaults to `true`.
- `-record/-r <true/t | false/f>`: Use `false` to prevent writing the schema changes made by the tool to the current **schema factory**. Defaults to `true`.
- `-schemas/-s <schema list>`: A list of schema and table names that Kodo should access during this run of the schematool. This is equivalent to setting the `kodo.jdbc.Schemas` property for a single run.

The schema tool also requires an `-action` flag. The available actions are:

- `add`: Bring the schema up-to-date with the given XML document by adding tables, columns, indexes, etc. This action never drops any schema components.
- `retain`: Keep all schema components in the given XML definition, but drop the rest from the database. This action never adds any schema components.
- `drop`: Drop all schema components in the schema XML. Tables will only be dropped if they would have 0 columns after dropping all columns listed in the XML.
- `refresh`: Equivalent to `retain`, then `add`.
- `createDB`: Generate SQL to re-create the current database. This action is typically used in conjunction with the `-file` flag to write a SQL script that can be used to create a fresh schema for a new database.
- `build`: Generate SQL to build a schema matching the one in the given XML file. Unlike `add`, this option does not take into account the fact that part of the schema defined in the XML file might already exist in the database. Therefore, this action is typically used in conjunction with the `-file` flag to write a SQL script. This script can later be used to recreate the schema in the XML on a fresh database.
- `dropDB`: Generate SQL to drop the current database. Like `createDB`, this action can be used with the `-file` flag to script a database drop rather than perform it.
- `import`: Import the given XML schema definition into the current schema factory. Does nothing if the factory does not store a record of the schema.
- `export`: Export the current schema factory's stored schema definition to XML. May produce an empty file if the factory does not store a record of the schema.

### Note

The schema tool can manipulate tables, columns, indexes, primary keys, and foreign keys. It cannot create or drop the database schema objects in which the tables reside, however. If your XML documents refer to named database schemas, those schemas must exist.

We present some examples of schema tool usage below.

### *Example 8.2. Schema Creation*

Add the necessary schema components to the database to match the given XML document, but don't drop any data:

```
schematool -a add targetSchema.xml
```

### Example 8.3. SQL Scripting

Repeat the same action as the first example, but this time don't change the database. Instead, write any planned changes to a SQL script:

```
schematool -a add -f script.txt targetSchema.xml
```

Write a SQL script that will re-create the current database:

```
schematool -a createDB -f script.txt
```

### Example 8.4. Schema Drop

Drop the current database:

```
schematool -a dropDB
```

## 8.3. XML Schema Format

The **schema generator**, **schema tool**, and **schema factories** all use the same XML format to represent database schema. The Document Type Definition (DTD) for schema information is presented below, followed by examples of schema definitions in XML.

```

<!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 auto-increment (true|false) "false">
<!ATTLIST column default CDATA #IMPLIED>
<!ATTLIST column size CDATA #IMPLIED>
<!ATTLIST column decimal-digits CDATA #IMPLIED>

<!-- the type-name attribute can be used when you want Kodo to -->
<!-- use a particular SQL type declaration when creating the -->
<!-- column. It is up to you to ensure that this type is -->
<!-- compatible with the JDBC type used in the type attribute. -->
<!ATTLIST column type-name 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">

<!-- the 'logical' attribute of pks should be set to 'true' if -->
<!-- the primary key does not actually exist in the database, -->
<!-- but the given column should be used as a primary key for -->
<!-- O-R purposes -->
<!ELEMENT pk (on)*>
<!ATTLIST pk name CDATA #IMPLIED>
<!ATTLIST pk column CDATA #IMPLIED>
<!ATTLIST pk logical (true|false) "false">

<!ELEMENT on EMPTY>
<!ATTLIST on column CDATA #REQUIRED>

<!-- fks with a delete-action of 'none' are similar to logical -->
<!-- pks; they do not actually exist in the database, but -->
<!-- represent a logical relation between tables (or their -->
<!-- corresponding Java classes) -->
<!ELEMENT fk (join)*>
<!ATTLIST fk name CDATA #IMPLIED>
<!ATTLIST fk deferred (true|false) "false">
<!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>
<!ATTLIST join value CDATA #IMPLIED>

```

### Example 8.5. Basic Schema

A very basic schema definition.

```

<schemas>
  <schema>
    <table name="ARTICLE">
      <column name="TITLE" type="varchar" size="255" not-null="true"/>
      <column name="AUTHOR_FNAME" type="varchar" size="28">
      <column name="AUTHOR_LNAME" type="varchar" size="28">
      <column name="CONTENT" type="clob">
    </table>
    <table name="AUTHOR">
      <column name="FIRST_NAME" type="varchar" size="28" not-null="true">
      <column name="LAST_NAME" type="varchar" size="28" not-null="true">
    </table>
  </schema>
</schemas>

```

### Example 8.6. Full Schema

Expansion of the above schema with primary keys, foreign keys, and indexes, some of which span multiple columns.

```

<schemas>
  <schema>
    <table name="ARTICLE">
      <column name="TITLE" type="varchar" size="255" not-null="true"/>
      <column name="AUTHOR_FNAME" type="varchar" size="28">
      <column name="AUTHOR_LNAME" type="varchar" size="28">
      <column name="CONTENT" type="clob">
      <pk column="TITLE"/>
      <fk to-table="AUTHOR" delete-action="exception">
        <join column="AUTHOR_FNAME" to-column="FIRST_NAME"/>
        <join column="AUTHOR_LNAME" to-column="LAST_NAME"/>
      </fk>
      <index name="ARTICLE_AUTHOR">

```

```

        <on column="AUTHOR_FNAME" />
        <on column="AUTHOR_LNAME" />
    </index>
</table>
<table name="AUTHOR">
    <column name="FIRST_NAME" type="varchar" size="28" not-null="true">
    <column name="LAST_NAME" type="varchar" size="28" not-null="true">
    <pk>
        <on column="FIRST_NAME" />
        <on column="LAST_NAME" />
    </pk>
</table>
</schema>
</schemas>

```

## 8.4. The SQLLine Utility

The Kodo distribution includes SQLLine, a console-based utility that interacts directly with a database using raw SQL. It is similar to other command-line database access utilities like `sqlplus` for Oracle, `mysql` for MySQL, and `isql` for Sybase/SQL Server. SQLLine can be useful as debugging tool by enabling low-level SQL interaction with the database.

SQLLine is open-source software. For complete documentation, see the project home page at <http://sqlline.sourceforge.net>. The remainder of this section discusses scenarios of using SQLLine to assist with understanding and developing Kodo applications.

### Note

As a separate utility, SolarMetric does not provide technical support for SQLLine.

### Example 8.7. Connecting to the Database

SQLLine can accept a `kodo.properties` file as a startup parameter, and will use it to connect to the database:

```

prompt$ java sqlline.SqlLine kodo.properties

Connecting to jdbc:hsqldb:tutorial_database
Connected to: HSQL Database Engine (version 1.7.0)
Driver: HSQL Database Engine Driver (version 1.7.0)
Autocommit status: true
sqlline version 0.7.8

0: jdbc:hsqldb:tutorial_database>

```

### Example 8.8. Examining the Tutorial Schema

SQLLine has various commands to analyze the schema of the database:

```
0: jdbc:hsqldb:tutorial_database> !tables
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_CA
		ANIMAL	TABLE		
		JDO_SEQUENCE	TABLE		

```
0: jdbc:hsqldb:tutorial_database> !columns ANIMAL
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_NAME	DATA_TYPE	TYPE_N
		ANIMAL	JDOCLASS	12	VARCHA
		ANIMAL	JDOID	-5	BIGINT
		ANIMAL	JDOVERSION	4	INTEGE
		ANIMAL	NAME0	12	VARCHA
		ANIMAL	PRICE	7	REAL

```
0: jdbc:hsqldb:tutorial_database> !primarykeys ANIMAL
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_NAME	KEY_SEQ	PK_NA
		ANIMAL	JDOID	1	SYS_PK_A

### Example 8.9. Issuing SQL Against the Database

Any SQL statement that the database understands can be executed in SQLLine, and the results (if any) will be displayed in a customizable format. The default is a table-like display:

```
0: jdbc:hsqldb:tutorial_database> SELECT * FROM ANIMAL;
```

JDOCLASS	JDOID	JDOVERSION	NAME0	PRICE
tutorial.Dog	0	0	Binney	80.0
tutorial.Dog	1	0	Fido	50.0
tutorial.Dog	2	0	Odie	30.0
tutorial.Dog	3	0	Tasha	75.0
tutorial.Dog	4	0	Rusty	25.0

```
5 rows selected (0 seconds)
```

```
0: jdbc:hsqldb:tutorial_database> DELETE FROM ANIMAL WHERE JDOID > 2;
```

```
2 rows affected (0.002 seconds)
```

```
0: jdbc:hsqldb:tutorial_database> SELECT * FROM ANIMAL;
```

JDOCLASS	JDOID	JDOVERSION	NAME0	PRICE
tutorial.Dog	0	0	Binney	80.0
tutorial.Dog	1	0	Fido	50.0
tutorial.Dog	2	0	Odie	30.0

```
3 rows selected (0 seconds)
```

```
0: jdbc:hsqldb:tutorial_database>
```

---

# Chapter 9. Runtime Deployment

Kodo JDO offers many deployment options.

## 9.1. JDOHelper

---

Kodo fully supports the `JDOHelper.getPersistenceManagerFactory` method, which is the standard way to obtain a persistence manager factory in JDO. This method is described in detail in the [JDO Overview](#). Just remember to include the `javax.jdo.PersistenceManagerFactoryClass` property to in the properties object you pass to the `JDOHelper`. The value of the property should be the full name of Kodo JDO's `JDBCPersistenceManagerFactory` class, so that the `JDOHelper` knows which factory implementation to instantiate.

### *Example 9.1. Specifying the PersistenceManagerFactory*

```
javax.jdo.PersistenceManagerFactoryClass: kodo.jdbc.runtime.JDBCPersistenceManagerFactory
```

The example below shows typical code to obtain a persistence manager factory from the `JDOHelper` using properties loaded from a file.

### *Example 9.2. Using the JDOHelper*

```
import java.io.*;
import java.util.*;
import javax.jdo.*;

...

Properties props = new Properties ();
InputStream propertyRes = getClass ().getClassLoader ().
    getResourceAsStream ("kodo.properties");
props.load (propertyRes);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory (props);
```

Once you have obtained a `PersistenceManagerFactory`, you can cache it for all persistence manager lookups. Even if you do not cache it yourself, Kodo JDO automatically pools persistence manager factories, so that subsequent calls to the `JDOHelper` with the same set of configuration properties will return the same factory instance.

## 9.2. KodoHelper

---

The `kodo.runtime.KodoHelper` is a static helper class much like `JDOHelper`. `KodoHelper`, however, contains many convenience methods that `JDOHelper` does not have. With the `KodoHelper`, you can obtain a persistence manager factory from a file, resource name, stream, or JNDI location. See its [Javadoc](#) for details.

### *Example 9.3. Using the KodoHelper*

```
import javax.jdo.*;
import kodo.runtime.*;

...

PersistenceManagerFactory pmf = KodoHelper.
    getPersistenceManagerFactory ("kodo.properties");
```

## 9.3. J2EE Deployment

---

Kodo JDO can be deployed through JCA in any JCA-compliant application server. For information on configuring and deploying Kodo in an application server environment, see [Section 4.3, “Installing Kodo JCA” \[103\]](#) in the J2EE Tutorial.

---

# Chapter 10. JDO Runtime Extensions

This chapter describes Kodo extensions to the standard JDO runtime interfaces, and outlines some additional features of the Kodo JDO runtime.

## 10.1. KodoPersistenceManagerFactory

---

The `kodo.runtime.KodoPersistenceManagerFactory` interface extends the basic `javax.jdo.PersistenceManagerFactory` with Kodo-specific features. The interface offers APIs to obtain managed and unmanaged persistence managers from the same factory and to perform other Kodo-specific operations. See the **interface Javadoc** for details on the `KodoPersistenceManagerFactory`.

## 10.2. KodoPersistenceManager

---

All Kodo persistence managers implement the `kodo.runtime.KodoPersistenceManager` interface. This interface extends the standard `javax.jdo.PersistenceManager`, and, just as the standard persistence manager is the primary window into JDO runtime services, the `KodoPersistenceManager` is the primary window into Kodo-specific functionality. We strongly encourage you to investigate the API extensions this interface contains, including many JDO 2.0 preview features.

### 10.2.1. JDO Transaction Events

---

One very important aspect of the `KodoPersistenceManager` is its support for broadcasting transaction-related events. By registering one or more `kodo.event.TransactionListener`'s with the persistence manager, you can receive notifications when transactions begin, flush, rollback, commit, and more. Where appropriate, event notifications include the set of persistence-capable objects participating in the transaction.

For details on the transaction framework, see the **Javadoc** for the `kodo.event` package.

#### Note

The Kodo JDO Performance Pack also supports **distributed events**.

### 10.2.2. JDO 2 Preview Methods

---

Kodo extends `PersistenceManager` with a variety of methods which preview JDO 2 features. These methods currently require casting to a `KodoPersistenceManager` to use. The **Javadoc** for details specifics with regards to using these methods.

#### Note

These methods are JDO 2 preview features and may change before the JDO 2 specification is finalized.

- `getObjectsById (Object[] oids, boolean validate)` : Return the instances with the corresponding object ids.
- `getObjectById (Class cls, Object value)`: This method returns the persistent instances of the given type by a variety of values:
  - Primary key value for single field identity or application identity with a single primary key field.

- Object id instances.
- `java.lang.Number` instances which wrap `kodo.util.Id` datastore primary key value. This is a Kodo extension of JDO functionality.
- Stringified id instances. This is a Kodo extension of JDO functionality.
- `attach/detach(All)` : Manage persistent copies which can be used to persist changes made outside of a `PersistenceManager` context, such as during remote use. See [Section 11.1, “Detach and Attach” \[297\]](#)
- `add/removeLifecycleListener (InstanceLifecycleListener listener, Class[] classes)` : Manage listeners which can be notified of events on persistent instances. See [Section 10.2.3, “Lifecycle Events” \[289\]](#)
- `refreshAll (JDOException)` : Refresh all failed objects contained in this exception, as well as any nested exceptions.
- `flush ()` : Flush changes on the current transactional instances to the datastore. This will trigger changes that would be executed on commit, without permanently altering the datastore.
- `checkConsistency ()` : Check the consistency of the persistence manager cache, including optimistic violations, constraint violations, etc.
- `put/remove/getUserObject (Object key, Object val)`: Manage key-value pairs in the map of user objects.
- `setRollbackOnly ()` : Will be moved to `javax.jdo.Transaction`. Defer transaction rollback until completion.

---

### 10.2.3. Lifecycle Events

Kodo includes the JDO 2 preview feature of listening for lifecycle events without implementing `InstanceCallbacks` on persistent classes. You can listen to state changes for the objects managed by a single persistence manager by adding **InstanceLifecycleListeners** to the `KodoPersistenceManager` of interest. Or, listen to state changes on all instances of all persistence managers by adding your listeners to the `KodoPersistenceManagerFactory`.

For details on the lifecycle event framework, see the **Javadoc** for the `kodo.event` package. For details on instance state changes, see [Section 4.4, “InstanceCallbacks” \[20\]](#).

#### Note

Lifecycle listeners are a JDO 2 preview feature and may change before the JDO 2 specification is finalized.

---

### 10.2.4. PersistenceManager Extension

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

Kodo JDO permits simple extension of the `PersistenceManager` used by the runtime. 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`, set the `kodo.PersistenceManagerImpl` configuration property to the full class name of your custom persistence manager.

As a **plugin string**, this property can also be used to configure persistence managers. All `PersistenceManagerImpl`s recognize the following properties:

- `CloseOnManagedCommit`: If `true`, then the persistence manager will be closed after a managed transaction (such as an EJB container-managed transaction) commits, assuming you have invoked the `close` method. If this is set to `false`, then the persistence manager will not be closed. This means that objects that were not properly detached from the persistence manager at the end of a session bean method and were then passed to a processing tier in the same JVM will still be usable, as their owning persistence manager will still be open. This behavior is not in strict compliance with the JDO specification, but is convenient for applications that were coded against Kodo 2, which did not close the persistence manager in these situations. The default for this property is `true`, meaning that the `PersistenceManager` will be properly closed.
- `EvictFromDataCache`: When evicting an object through the standard persistence manager evict methods, whether to also evict it from the Kodo's **datastore cache**. Defaults to `false`.

---

## 10.3. KodoExtent

Kodo JDO extends the base `javax.jdo.Extent` with the `kodo.runtime.KodoExtent`. The `KodoExtent` offers many convenience methods and object-loading configuration options through its **fetch configuration**.

---

## 10.4. KodoQuery

Kodo queries provide many configuration parameters and functions beyond those of the standard `javax.jdo.Query` interface defined in JDO 1. For example, Kodo supports many JDO 2 preview features, such as aggregates, projections, grouping, having, custom result classes, and result ranges. SolarMetric's JDO Overview includes a discussion of these JDO 2 query features in [Chapter 11, \*Query\*](#) [54]. Access to JDO 2 preview APIs and other extended Kodo functionality is available through the `kodo.query.KodoQuery` interface.

---

## 10.5. Fetch Configuration

Many of the aforementioned Kodo interfaces give you access to a `kodo.runtime.FetchConfiguration` instance. The `FetchConfiguration` allows you to exercise some control over how objects are fetched from the data store, including **large result set support**, **custom fetch groups**, and **lock levels**. You can cast any fetch configuration Kodo returns to its JDBC-specific subclass, the `kodo.jdbc.runtime.JDBCFetchConfiguration`. See its **Javadoc** for details.

Fetch configurations are passed on from parent components to child components. The persistence manager factory settings (via the configuration properties) for things like the fetch batch size, result set type, and custom fetch groups are passed on to the fetch configuration of the persistence managers it produces. The settings of each persistence manager, in turn, are passed on to all queries and extents it returns. Note that the opposite, however, is not true. Modifying the fetch configuration of a query or extent does not affect the persistence manager's configuration. Likewise, modifying a persistence manager's configuration does not affect the persistence manager factory.

---

## 10.6. KodoHelper

The `kodo.runtime.KodoHelper` is a static helper class, much like `JDOHelper`. `KodoHelper` provides many convenience methods for obtaining persistence manager factories, as well as methods to provide Kodo-specific information about persistent instances, such as their JDO metadata objects or state managers. Finally, the static `KodoHelper.close` method can be used to close any Kodo resource, such as an extent iterator, query result, or **large result set field** iterator, without needing a reference to the owning extent, query, or object. See `KodoHelper`'s **Javadoc** for details.

---

## 10.7. 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 provides alternatives and extensions to standard JDO queries. This section discusses JDOQL query extensions. For a discussion of **subqueries**, **direct SQL queries** and **custom query execution**, see [Chapter 13, \*Enterprise Edition\*](#) [321]

---

## 10.7.1. JDOQL Extensions

---

JDOQL extensions are custom methods that you can use in your query filter, having, ordering, and result strings. Kodo JDO provides some built-in JDOQL extensions, and you can develop your own custom extensions as needed. You can optionally preface all JDOQL extensions with `ext:` in your JDOQL query. For example, the following example uses a hypothetical `firstThreeChars` extension to search for cities whose name begins with the 3 characters 'H', 'a', 'r'.

### *Example 10.1. Basic JDOQL Extension*

```
Query query = pm.newQuery (City.class);
query.setFilter ("name.ext:firstThreeChars () == 'Har'");
Collection results = (Collection) query.execute ();
```

Note that it is perfectly OK to chain together extensions. For example, let's modify our search above to be case-insensitive using another hypothetical extension, `equalsIgnoreCase`:

### *Example 10.2. Chaining JDOQL Extensions*

```
Query query = pm.newQuery (City.class);
query.setFilter ("name.ext:firstThreeChars ().ext>equalsIgnoreCase ('Har')");
Collection c = (Collection) query.execute ();
```

Finally, when using JDOQL extensions you must be aware that any 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).

### 10.7.1.1. Included JDOQL Extensions

---

Kodo includes two default JDOQL extensions to enhance the power of JDOQL. Note that Kodo natively supports all of the JDOQL methods defined in the JDO 2 early draft specification, so those methods are not listed here as extensions. See [Chapter 11, \*Query\* \[54\]](#) for a primer on JDO 2 queries.

- `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'");
```

- `sql`: Embeds the given SQL argument into the `SELECT` statement that is issued. This filter cannot be used for in-memory queries.

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

This extension replaces `sqlVal` and `sqlExp`, which are now deprecated.

### 10.7.1.2. Developing Custom JDOQL Extensions

---

You can write your own extensions by implementing the `kodo.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 `kodo.query` and `kodo.jdbc.query` packages.

### 10.7.1.3. Configuring JDOQL Extensions

---

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

- *Registration by properties:* You can register custom JDOQL extensions by setting the `kodo.FilterListeners` configuration property to a comma-separated list of **plugin strings** describing 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 JDOQL extensions for an individual query through the `KodoQuery.addFilterListener` method. You might use per-query registration for very specific extensions that do not apply globally.

## 10.7.2. Aggregate Extensions

---

Just as you can write your own JDOQL methods, you can write your own query aggregates by implementing the `kodo.jdbc.query.JDBCAggregateListener` interface. View the Javadoc documentation for details. When using your custom aggregates in result or having strings, you can optionally prefix the function name with `ext :` to identify it as an extension.

### 10.7.2.1. Configuring Query Aggregates

---

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

- *Registration by properties:* You can register custom query aggregates by setting the `kodo.AggregateListeners` configuration property to a comma-separated list of **plugin strings** describing your aggregate implementation. Aggregates 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 aggregates for an individual query through the `KodoQuery.addAggregateListener` method. You might use per-query registration for very specific aggregates that do not apply globally.

## 10.8. Object Locking

---

Controlling how and when objects are locked is an important part of maximizing the performance of your application under load. This section describes Kodo's APIs for explicit locking, as well as its rules for implicit locking.

---

## 10.8.1. Configuring Default Locking

You can control Kodo's default transactional read and write lock levels through the `kodo.ReadLockLevel` and `kodo.WriteLockLevel` configuration properties. Each property accepts a value of `none`, `read`, `write`, or a number corresponding to a lock level defined by the **lock manager** in use. These properties apply only to non-optimistic transactions; during optimistic transactions, Kodo never locks objects by default.

You can control the default amount of time Kodo will wait when trying to obtain locks through the `kodo.LockTimeout` configuration property. Set this property to the number of milliseconds you are willing to wait for a lock before Kodo will throw an exception, or to -1 for no limit. It defaults to -1.

### *Example 10.3. Setting Default Lock Levels*

```
kodo.ReadLockLevel: none
kodo.WriteLockLevel: write
kodo.LockTimeout: 30000
```

## 10.8.2. Configuring Lock Levels at Runtime

At runtime, you can override the default lock levels through the `FetchConfiguration` interface, described in [Section 10.5, “Fetch Configuration”](#) [290]. At the beginning of each datastore transaction, Kodo initializes the persistence manager's fetch configuration with the default lock levels and timeouts described in the previous section. By changing the fetch configuration's locking properties, you can control how objects loaded at different points in the transaction are locked. You can also use the fetch configuration of an individual query or extent to apply your locking changes only to objects loaded through that query or extent.

Controlling locking through the fetch configuration works even during optimistic transactions. At the end of the transaction, Kodo resets the fetch configuration's lock levels to `none`. You cannot lock objects outside of a transaction.

### *Example 10.4. Setting Runtime Lock Levels*

```
pm.currentTransaction ().begin ();

// load stock we know we're going to update at write lock level
KodoQuery kq = (KodoQuery) pm.newQuery (Stock.class, "symbol == :s");
kq.setUnique (true);
FetchConfiguration fetch = kq.getFetchConfiguration ();
fetch.setReadLockLevel (fetch.LOCK_WRITE);
fetch.setLockTimeout (3000); // 3 seconds
Stock stock = (Stock) kq.execute (symbol);

// load an object we don't need locked at none lock level
fetch = ((KodoPersistenceManager) pm).getFetchConfiguration ();
fetch.setReadLockLevel (fetch.LOCK_NONE);
Market market = (Market) pm.getObjectById (marketId, false);

stock.setPrice (market.calculatePrice (stock));
pm.currentTransaction ().commit ();
```

## 10.8.3. Object Locking APIs

In addition to allowing you to control implicit locking levels, Kodo provides explicit APIs to lock objects and to retrieve their current lock level:

- `KodoHelper.getLockLevel(Object obj)`: Return the numeric level at which the given object is currently locked.
- `KodoPersistenceManager.lockPersistent(Object obj, int level, long timeout)`: Lock the given object at the specified lock level and timeout. The level is one of the `LOCK_XXX` constants defined in `KodoPersistenceManager` (and mirrored in `FetchConfiguration`), or a numeric value corresponding to a recognized lock level of the **lock manager** in use.
- `KodoPersistenceManager.lockPersistent(Object obj)`: Same as above, but using the fetch configuration's current write lock level and lock timeout.
- `KodoPersistenceManager.lockPersistentAll(Collection objs, int level, long timeout)`: Same as `lockPersistent`, but locks all of the given objects. The given collection may contain nulls, but may not be null itself.
- `KodoPersistenceManager.lockPersistentAll(Collection objs)`: Same as above, but using the fetch configuration's current write lock level and lock timeout.
- `KodoPersistenceManager.lockPersistentAll(Object[] objs, int level, long timeout)`: Same as `lockPersistent`, but locks all of the given objects. The given array may contain nulls, but may not be null itself.
- `KodoPersistenceManager.lockPersistentAll(Object[] objs)`: Same as above, but using the fetch configuration's current write lock level and lock timeout.

### *Example 10.5. Using `lockPersistent()`*

```
pm.currentTransaction ().setOptimistic (true);
pm.currentTransaction ().begin ();

// override default of not locking during an opt trans to lock stock object
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.lockPersistent (stock, kpm.LOCK_WRITE, -1);
stock.setPrice (market.calculatePrice (stock));

pm.currentTransaction ().commit ();
```

---

## 10.8.4. Lock Manager

Kodo delegates the actual work of locking objects to the system's **`kodo.runtime.LockManager`**. This plugin is controlled by the **`kodo.LockManager`** configuration property. You can write your own lock manager, or use one of the bundled options:

- `pessimistic`: The default. This is an alias for the **`kodo.jdbc.runtime.PessimisticLockManager`**, which uses `SELECT FOR UPDATE` statements (or the database's equivalent) to lock the database rows corresponding to locked objects. This lock manager does not distinguish between read locks and write locks; all locks are write locks.
- `none`: An alias for the **`kodo.runtime.NoLockManager`**, which does not perform any locking at all.
- `sjvm`: An alias for the **`kodo.runtime.SingleJVMExclusiveLockManager`**. This lock manager uses in-memory mutexes to obtain exclusive locks on object ids. It does not perform any database-level locking. Also, it does not distinguish between read and write locks; all locks are write locks.

### *Example 10.6. Disabling Locking*

```
kodo.LockManager: none
```

## 10.8.5. Rules for Locking Behavior

---

JDO's advanced concepts like lazy-loading and object uniquing create several locking corner-cases. The rules below outline Kodo's implicit locking behavior in these cases.

1. When an object's state is first read within a transaction, the object is locked at the fetch configuration's current read lock level. Future reads of additional lazy state for the object will use the same read lock level, even if the fetch configuration's level has changed.
2. When an object's state is first modified within a transaction, the object is locked at the write lock level in effect when the object was first read, even if the fetch configuration's level has changed. If the object was not read previously, the current write lock level is used.
3. When objects are accessed through a persistent relation field, the related objects are loaded with the fetch configuration's current lock levels, not the lock levels of the object owning the field.
4. Whenever an object is looked up within a transaction (through a query, extent, `getObjectById`, or previously-unloaded relation), the object is re-locked at the current read lock level. The current read and write lock levels become those that the object "remembers" according to rules one and two above.
5. If you lock an object explicitly through the `lockPersistent` methods, it is re-locked at the specified level. This level also becomes both the read and write level that the object "remembers" according to rules one and two above.
6. When an object is already locked at a given lock level, re-locking at a lower level has no effect. Locks cannot be downgraded during a transaction.

## 10.8.6. Known Issues and Limitations

---

Due to performance concerns and database limitations, locking cannot be perfect. You should be aware of the issues outlined in this section, as they may affect your application.

- Typically, during optimistic JDO transactions Kodo does not start an actual database transaction until you flush or the optimistic transaction commits. This allows for very long-lived transactions without consuming database resources. When using the default lock manager, however, Kodo must begin a database transaction whenever you decide to lock an object during an optimistic JDO transaction. This is because the default lock manager uses database locks, and databases cannot lock rows without a transaction in progress. Kodo will log an INFO message to the `kodo.Runtime` logging channel when it begins a datastore transaction just to lock an object.
- In order to maintain reasonable performance levels when loading object state, Kodo can only guarantee that an object is locked at the proper lock level *after* the state has been retrieved from the database. This means that it is technically possible for another transaction to "sneak in" and modify the database record after Kodo retrieves the state, but before it locks the object. The only way to positively guarantee that the object is locked and has the most recent state to refresh the object after locking it.

When using the default lock manager, the case above can only occur when Kodo cannot issue the state-loading SELECT as a locking statement due to database limitations. For example, some databases cannot lock SELECTs that use joins. The default lock manager will log an INFO message to the `kodo.Runtime` logging channel whenever it cannot lock the initial SELECT due to database limitations. By paying attention to these log messages, you can see where you might consider using an object refresh to guarantee that you have the most recent state, or where you might rethink the way you load the state in ques-

tion to circumvent the database limitations that prevent Kodo from issuing a locking `SELECT` in the first place.

## 10.9. Orphaned Keys

---

Unless you apply database foreign key constraints extensively, it is possible to end up with orphaned keys in your database. For example, suppose `Person p` has a reference to `Address a`. If you delete `a` without nulling `p`'s reference, `p`'s database record will wind up with an orphaned key to the non-existent `a` record.

### Note

One way of avoiding orphaned keys is to use *dependent* fields. See [Section 6.2.1.3, “dependent” \[198\]](#).

Kodo's `kodo.OrphanedKeyAction` configuration property controls what action to take when Kodo encounters an orphaned key. You can set this plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) to a custom implementation of the `kodo.event.OrphanedKeyAction` interface, or use one of the built-in options:

- `log`: Log a message for each orphaned key. This is an alias for the `kodo.event.LogOrphanedKeyAction` class. This setting has the following additional properties:
  - `Channel`: The channel to log to. Defaults to `kodo.Runtime`.
  - `Level`: The level to log at. Defaults to `WARN`.
- `exception`: Throw a `javax.jdo.JDOObjectNotFoundException` when Kodo discovers an orphaned key. This is an alias for the `kodo.event.ExceptionOrphanedKeyAction` class.
- `none`: Ignore orphaned keys. This is an alias for the `kodo.event.NoOrphanedKeyAction` class.

### Example 10.7. Custom Logging Orphaned Keys

```
kodo.OrphanedKeyAction: log(Channel=Orphans, Level=DEBUG)
```

---

# Chapter 11. Remote and Offline JDO

The standard JDO runtime environment is *local* and *online*. It is *local* in that JDO components such as persistence managers and queries connect directly to the datastore and execute their actions in the same JVM as the code using them. It is *online* in that all changes to persistent objects must be made in the context of an active persistence manager. These two properties, combined with the fact that persistence managers cannot be serialized for storage or network transfer, make the standard JDO runtime difficult to incorporate into some enterprise and client/server program designs.

Kodo extends the standard JDO runtime to add *remote* and *offline* capabilities in the form of **Remote Persistence Managers** and **Detach and Attach APIs**. The following sections explain these capabilities in detail.

## 11.1. Detach and Attach

---

A common use case for an application running in a servlet or application server is to "detach" objects from all server resources, modify them, and then "attach" them again. For example, a servlet might store persistent data in a user session between a modification based on a series of web forms. Between each form request, the web container might decide to serialize the session, requiring that the stored persistent state be disassociated from any other resources. Similarly, a client/server or EJB application might transfer persistent objects to a client via serialization, allow the client to modify their state, and then have the client return the modified data in order to be saved. This is sometimes referred to as the "data transfer object" or "value object" pattern, and it allows fine-grained manipulation of data objects without incurring the overhead of multiple remote method invocations (which is one of the things that often makes entity bean-based solutions slow). Version 1 of the JDO specification does not provide direct support for this pattern or any offline modification, since persistent object updates can only take place when the instance is associated with a persistence manager that has a transaction running.

Kodo provides support for this pattern by introducing *detach* and *attach* APIs that allow a user to detach a persistent instance, modify the detached instance, and attach the instance back into a persistence manager (either the same one that detached the instance, or a new one). The changes will then be applied to the existing instance from the datastore. Very similar APIs are being considered for inclusion in the JDO 2 specification, so they may become standard in the future.

### 11.1.1. Declaring Detachability

---

In order to be able to detach a persistent instance, the metadata for the class must declare that it is eligible for detachment using the `detachable` extension. Changes to this extension require that the class be re-enhanced. This is because detachability requires that the enhancer add additional fields to the class to hold information about the persistent instance's object id and state.

#### Note

Kodo uses reflection to access the detached state, so any detachable classes must be declared `public`.

The simplest example of the detachable extension is:

```
public class DetachExample
{
    private String someField;
}

<?xml version="1.0"?>
<jdo>
  <package name="com.somecompany">
    <class name="DetachExample">
      <extension vendor-name="kodo" key="detachable" value="true"/>
    </class>
  </package>
</jdo>
```

As mentioned previously, when a class is declared to be detachable, the Kodo enhancer adds additional fields to the enhanced version of the class. One of these fields is of type `Object`, and holds an object that refers to the class' state. Kodo uses this field for bookkeeping information, including the versioning data needed to detect optimistic concurrency violations when the object is re-attached. If the persistent class uses datastore identity, the enhancer adds a second `String` field used to store the stringified object id of the detached instance.

It is possible to define one or both of these fields yourself. Declaring your own state and identity fields in your class metadata prevents the enhancer from adding any extra fields to the class, and keeps the enhanced class serialization-compatible with the unenhanced version (in case the client tier only has the unenhanced class definition to work with). The `detached-object-id-field` and `detached-state-field` class-level metadata extensions name these fields. The fields must not be managed by JDO (they must have their `persistence-modifier` attribute set to `"none"`).

```
public class DetachExample
    implements Serializable
{
    private String someField;
    private String detachedObjectId;
    private Object detachedState;
}

<?xml version="1.0"?>
<jdo>
  <package name="com.somecompany">
    <class name="DetachExample">
      <extension vendor-name="kodo" key="detachable" value="true"/>
      <extension vendor-name="kodo" key="detached-objectid-field"
        value="detachedObjectId"/>
      <extension vendor-name="kodo" key="detached-state-field"
        value="detachedState"/>

      <!-- string fields are normally managed by default, so explicitly set -->
      <!-- this field to unmanaged; we don't need to worry about the -->
      <!-- detachedState fields because fields of type java.lang.Object -->
      <!-- are not managed by default -->
      <field name="detachedObjectId" persistence-modifier="none"/>
    </class>
  </package>
</jdo>
```

---

## 11.1.2. Detach and Attach Behavior

The `KodoPersistenceManager` exposes three methods to detach objects:

- `Object detach(Object ob)`
- `Object[] detachAll(Object[] obs)`
- `Collection detachAll(Collection obs)`

Each detach method returns unmanaged copies of the given instances. The copy mechanism is similar to serialization, except that only certain fields are traversed. We will see how to control which fields are detached in a later section.

When detaching an instance that has been modified in the current transaction (and thus made dirty), the current transaction is flushed. This means that when subsequently re-attaching the detached instances, Kodo assumes that the transaction from which they were originally detached was committed; if it has been rolled back, then the re-attachment process will throw a `JDOOptimisticVerificationException`. You can stop Kodo from assuming the transaction will commit by invoking `KodoPersistenceManager.setRollbackOnly` prior to detaching your objects. Setting the `RollbackOnly` flag prevents Kodo from flushing when detaching dirty objects; instead Kodo just runs its pre-store actions (see the `KodoPersistenceManager.preStore` **Javadoc** for details). This allows you to use the same instances in multiple attach/modify/detach/rollback cycles. Alternatively, you might also prevent a flush by making your modifications outside of a transaction (with `NontransactionalWrite` enabled) before detaching.

For every detach method, there is a corresponding attach version:

- `Object attach(Object ob)`
- `Object[] attachAll(Object[] obs)`
- `Collection attachAll(Collection obs)`

Each attach method returns managed copies of the given detached objects. Any changes made to the detached objects are applied to these managed instances. Because attaching involves changing persistent state, you can only attach within a transaction.

If you attempt to attach an instance whose representation has changed in the datastore since detachment, Kodo will throw a `JDOOptimisticVerificationException` upon commit or flush, just as if a normal optimistic conflict was detected. When attaching an instance whose database record has been deleted since detaching, or when attaching a detached instance into a persistence manager that has a stale version of the object, Kodo will throw a `JDOOptimisticVerificationException` from the attach method. In these cases, Kodo sets the `RollbackOnly` flag on the transaction.

### ***Example 11.1. Detaching and Attaching a Single Instance***

This example demonstrates a common client/server scenario. The client requests objects and makes changes to them, while the server handles the object lookups and transactions.

```
import kodo.runtime.*;

...

// CLIENT:
// requests an object with a given oid
DetachExample detached = (DetachExample) getFromServer (oid);

...

// SERVER:
// detaches object and returns to client
Object oid = processClientRequest ();
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
DetachExample example = (DetachExample) kpm.getObjectById (oid, false);
DetachExample detached = (DetachExample) kpm.detach (example);
sendToClient (detached);

...

// CLIENT:
// makes some modifications and sends back to server
detached.setSomeField ("bar");
sendToServer (detached);

...

// SERVER:
// re-attaches the instance and commit the changes
DetachExample modified = (DetachExample) processClientRequest ();
kpm.currentTransaction ().begin ();
kpm.attach (modified);
kpm.currentTransaction ().commit ();
```

---

## **11.1.3. Defining the Detached Object Graph**

After viewing the example above, you may be wondering exactly what state of the object being transferred back and forth is available to the client. Can the client traverse into relations? If so, how deep? Kodo supports three modes for determining which fields and relations of an object are detached. All of the modes are recursive.

1. `KodoPersistenceManager.DETACH_FGS`: Detach all fields and relations in the default fetch group, and any other fetch groups that you have added to the persistence manager's **fetch configuration**. This is the default mode. For more information on custom fetch groups, see **Section 14.5, “Fetch Groups”** [337]
2. `KodoPersistenceManager.DETACH_LOADED`: Detach all fields and relations that are already loaded, but don't include unloaded fields in the detached graph.
3. `KodoPersistenceManager.DETACH_ALL`: Detach all fields and relations. Be very careful when using this mode; if you have a highly-connected domain model, you could end up bringing every object in the database into memory!

Any field that is not included in the set determined by the detach mode is set to its Java default value in the detached instance. You can set the detach mode at any time using the `KodoPersistenceManager.setDetachFields(int)` method:

```
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.setDetachFields (kpm.DETACH_LOADED);
```

You can also change the mode persistence managers are initialized with using the `kodo.PersistenceManagerImpl` configuration property:

```
# available settings are "fgs", "loaded", "all"
kodo.PersistenceManagerImpl: DetachFields=loaded
```

Here is an example using the default detach mode with a custom fetch group to control the detached graph declaratively.

### ***Example 11.2. Using Custom Fetch Groups for Detach***

```
public class DetachExample
    implements Serializable
{
    private String someField;
    private DetachExampleRelation someRelation;
}

<?xml version="1.0"?>
<jdo>
  <package name="com.somecompany">
    <class name="DetachExample">
      <extension vendor-name="kodo" key="detachable" value="true"/>
      <field name="someRelation">
        <extension vendor-name="kodo" key="fetch-group" value="mygroup"/>
      </field>
    </class>
    <class name="DetachExampleRelation">
      <extension vendor-name="kodo" key="detachable" value="true"/>
    </class>
  </package>
</jdo>

import kodo.runtime.*;

...

// CLIENT:
// requests an object with a given oid
DetachExample detached = (DetachExample) getFromServer (oid);

...

// SERVER:
// detaches object and returns to client
Object oid = processClientRequest ();
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getFetchConfiguration ().addFetchGroup ("mygroup");
```

```
DetachExample example = (DetachExample) kpm.getObjectById (oid, false);
DetachExample detached = (DetachExample) kpm.detach (example);
sendToClient (detached);

...

// CLIENT:
// makes some modifications and sends back to server
detached.setSomeField ("bar");
detached.getRelation ().setSomeOtherField ("baz");
sendToServer (detached);

...

// SERVER:
// re-attaches the instance and commit the changes
DetachExample modified = (DetachExample) processClientRequest ();
kpm.currentTransaction ().begin ();
kpm.attach (modified); // will recursively attach relation and apply mods
kpm.currentTransaction ().commit ();
```

---

## 11.1.4. Detach and Attach Callbacks

Persistent instances can be notified when they are being detached or attached by implementing the **`kodo.runtime.PreDetachCallback`** and **`kodo.runtime.PostDetachCallback`** interfaces. These interfaces are analogous to the notification mechanisms provided by the `javax.jdo.InstanceCallbacks` interface (see [Section 4.4](#), “**InstanceCallbacks**” [20]).

- `PreDetachCallback.jdoPreDetach` is called on the managed persistent instance before it be to be detached.
- `PostDetachCallback.jdoPostDetach` is called on the detached copy of the persistent instance after the entire detach process is complete. The method takes a single `java.lang.Object` argument, which is the managed instance that the detached copy is the clone of. It can be used to transfer transient or unmanaged state between the managed instance and the detached instance.
- `PreAttachCallback.jdoPreAttach` is called on a detached instance before it is to be attached.
- `PostAttachCallback.jdoPostAttach` is called on the managed instance after it has been attached. It is only invoked after the entire attach process is complete. The method takes a single `java.lang.Object` argument, which is the corresponding detached instance. It can be used to transfer transient or unmanaged state between the detached instance and the attached instance.

---

## 11.1.5. Automatic Detachment

Automatic detachment encompasses two features: detach-on-close and detach-on-serialize. Each feature is designed to simplify certain patterns of enterprise application development.

---

### 11.1.5.1. Detach on Close

Normally when you close a persistence manager, all of its managed objects become invalid. Attempting to manipulate these objects results in undefined behavior; often you will get an exception. With the detach-on-close feature, however, you can configure persistence managers to automatically detach all of their objects when they close. This allows you to continue working with the objects in their detached state, and to later re-attach them to apply any changes you might make. Detach on close is designed for the following scenarios in particular:

- **Client/Server applications with short-lived persistence managers.** A common server architecture handles each request by obtaining a persistence manager, looking up persistent objects, closing the persistence manager, and returning the objects to

the client. Using detach-on-close ensures that persistent objects remain valid for return without forcing you to detach them explicitly.

- **Servlets and JSPs.** A very common pattern in servlet and JSP architectures is to allocate a persistence manager at the beginning of a web request, then close it at the end of the request. The objects used by the persistence manager during the request are usually discarded. Sometimes, though, you need to store persistent objects in the HTTP session for use in later requests. With detach-on-close, you no longer need to explicitly detach the objects you want to retain. Simply place persistent objects into the session, and when you close the persistence manager at the end of the request the objects will detach transparently.
- **Session Beans.** Each business method of a session bean is supposed to obtain a persistence manager, use it for the necessary persistence operations, then close it. But you often need to return persistent objects from your business methods, and closing the persistence manager invalidates the objects you're trying to return! The standard workaround is to use `makeTransient` or `detach` to separate the return objects from the persistence manager. With detach-on-close, though, this step is no longer needed. If you configure the persistence manager to detach all of its objects when it closes, the return objects will remain valid. The code calling your session bean can access their state, or even modify them and send them back to your bean for re-attachment.

### Note

In transactional enterprise bean methods, calling `PersistenceManager.close` may not actually close the persistence manager. Rather, the persistence manager stays open until the managed transaction in progress completes. Thus, your return objects may be serialized before the detach-on-close feature is activated. In transactional EJB code, therefore, you should combine detach-on-close with detach-on-serialize, discussed in [Section 11.1.5.2, “Detach on Serialize” \[302\]](#)

The detach-on-close feature is off by default. You can toggle it at runtime though the `KodoPersistenceManager.setDetachOnClose` method. You can also set the value new persistence managers are initialized with using the `kodo.PersistenceManagerImpl` configuration property:

```
kodo.PersistenceManagerImpl: DetachOnClose=true
```

---

## 11.1.5.2. Detach on Serialize

JDO serialization is designed to be transparent. When a managed object is serialized, it loads all of its persistent fields and relations so that the serialized graph contains the full state of the object. This is in keeping with standard Java serialization, but often it isn't the behavior you want. With a highly connected persistent graph, the recursive serialization process might end up pulling a large percentage of the entire database into memory!

Kodo's detach-on-serialize feature stresses practicality over strict transparency. A class that is configured to use detach on serialize writes a detached version of itself to the serialization stream. This has two significant benefits:

1. You have control over the size of the serialized graph. Rather than each object reading in its entire state, your detach mode determines which fields are serialized. See [Section 11.1.3, “Defining the Detached Object Graph” \[299\]](#).
2. The resulting deserialized objects are detached instances. They retain their persistent identity and version, so you can re-attach them later without creating new datastore records for them.

The detach-on-serialize feature is particularly useful in the following use cases:

- **Client/Server applications with long-lived persistence managers.** In a client/server architecture in which the server uses

one persistence manager per request, you should use **detach-on-close**, because the persistence manager will close before any return objects are serialized to the client. But if the server keeps persistence managers open for multiple requests, detach-on-serialize allows you to pass persistent objects over the wire from the server to the client without explicit detach calls.

- **Session Beans.** Each business method of a session bean should close the persistence manager at the end of the method. When there is a managed transaction in progress, however, the persistence manager cannot actually close until the transaction completes. Some application servers will serialize the return value of your business method before completing the transaction, meaning that the persistence manager is still open during serialization. Using detach-on-serialize ensures that the code calling your session bean receives detached copies of the return objects without any explicit detach calls. You should typically combine detach-on-serialize with detach-on-close, covered in **Section 11.1.5.1, “Detach on Close” [301]**.

Serializing objects to a detached version involves changes to how classes are enhanced. Therefore, the detach-on-serialize feature isn't controlled by a runtime flag or configuration setting. Instead, you tag classes that should serialize to a detached instance by setting the detachable metadata extension to `serialize` instead of `true`:

```
<?xml version="1.0"?>
<jdo>
  <package name="com.somecompany">
    <class name="DetachExample">
      <extension vendor-name="kodo" key="detachable" value="serialize"/>
    </class>
  </package>
</jdo>
```

### Note

The enhancer implements the `java.io.Externalizable` interface for classes that detach on serialize. This means that you cannot combine detach-on-serialize with your own custom serialization code.

Also, the detach callbacks discussed in **Section 11.1.4, “Detach and Attach Callbacks” [301]** are not performed on instances that are detached through serialization.

## 11.2. Remote Persistence Managers

---

In JDO, each persistence manager factory maintains a set of resources shared by all the persistence managers produced by that factory. Sharing common structures like connection pools and data caches drastically reduces the resource consumption of each persistence manager, increasing your application's scalability. Kodo takes this concept one step further by also giving its persistence manager factories the ability to act as servers for persistence managers on remote machines. You can thus leverage the full JDO API in your client tier without duplicating limited resources like database connections on each client. In addition to ensuring that your application scales as more clients are added, this model may allow you to use Kodo in situations where the client machine cannot directly access the necessary server-side resources itself - for example, when the database is only available to the local network.

This remote capability means that you can design your application as a simple two-tiered servlet-database application, and then migrate to a more scalable servlet-JDO middle tier-database architecture as the load on your system increases. This end picture looks much like a standard J2EE application server architecture, except that the code that uses the JDO APIs in the servlet does not need to change at all to toggle between the more performant two-tier architecture and the more scalable three-tier architecture.

Additionally, Kodo's remote capability is useful for applet and Java Web Start application development. In conjunction with the compressed HTTP transport, you can deploy code that uses JDO via an applet or a Web Start application. The persistence managers in these applications will then connect back to the server that they were downloaded from in order to access the database.

### Note

Kodo's remote persistence manager capabilities require an Enterprise Edition license in order to function. See [Chapter 13, \*Enterprise Edition\* \[321\]](#) for details about what is included in the Enterprise Edition.

## 11.2.1. Standalone Persistence Manager Server

---

To configure a persistence manager factory to act as a standalone server to remote persistence managers, set the factory's `kodo.PersistenceManagerServer` configuration property to a plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)) describing the `com.solarmetric.remote.Transport` implementation to use for remote communication. You can implement your own `Transport`, or use one of the built-in options:

- `false`: The default value. No server is started.
- `tcp`: An alias for `com.solarmetric.remote.TCPTransport`, a TCP transport layer. This transport layer has the following settings:
  - `Port`: The port the server will listen on. Defaults to 5637.
  - `Host`: The host name of the server. Defaults to `localhost`. This setting is not used by the server, but by clients. We discuss client configuration below.
  - `SoTimeout`: The socket read timeout in milliseconds. Defaults to 0 for no timeout.
  - `Decorators`: See [Section 11.2.4, “Data Compression and Filtering” \[307\]](#)

### *Example 11.3. Configuring a Standalone Persistence Manager Server*

```
kodo.PersistenceManagerServer: tcp(Port=5555)
```

When a persistence manager factory is configured as a standalone server, it begins listening for client connections automatically as soon as you obtain the factory. The server will run for as long as the factory remains open, and will shut down when you close the factory. The `kodo.runtime.KodoPersistenceManagerFactory` interface also provides methods for manually managing its server thread. See its [Javadoc](#) for details.

Your Kodo distribution includes a program to start a standalone persistence manager server. You can run the program through its Java class, `kodo.jdbc.runtime.StartPersistenceManagerServer`, or through the provided `startpmsserver` command-line script. The script accepts the standard Kodo command-line arguments outlined in [Section 2.3, “Command Line Configuration” \[139\]](#).

### *Example 11.4. Starting a Standalone Persistence Manager Server*

```
startpmsserver -p server.properties
```

## 11.2.2. HTTP Persistence Manager Server

---

Kodo's remote persistence managers can communicate with the server over HTTP, allowing you to use them through firewalls that shut off other ports and protocols. In order to receive HTTP requests from remote clients, Kodo includes the `kodo.remote.PersistenceManagerServerServlet`. You can deploy this servlet into any standards-compliant servlet container.

The `PersistenceManagerServerServlet` services remote persistence manager requests using an internal persistence manager factory. The servlet provides several mechanisms for configuring this persistence manager factory:

1. First, the servlet checks the value of the `kodo.jndi` servlet initialization parameter (servlet initialization parameters are specified in the standard `web.xml` deployment file; see your servlet container documentation for details). If the value of this parameter is non-null, Kodo attempts to look up the persistence manager factory at the indicated JNDI location.
2. If the `kodo.jndi` initialization parameter is not set, Kodo checks the `kodo.properties` initialization parameter. The value of this parameter is a resource path to a Java properties file containing Kodo configuration properties. If no value is given, the resource path defaults to `/kodo.properties`, which corresponds to a `kodo.properties` file located in the root of the WAR or expanded servlet directory.
3. Finally, Kodo checks the remainder of the servlet initialization parameters for Kodo configuration properties. These parameter values override the value supplied in the properties file (if any).

You can make sure that the servlet's persistence manager factory is configured as expected by navigating to the servlet in your web browser. The servlet will display a simple web page detailing the configuration of its internal factory.

## 11.2.3. Client Persistence Managers

---

Client persistence managers are remote proxies to server-side persistence managers created by the server-side persistence manager factory you are communicating with. From an API standpoint, a client persistence manager is exactly like a local one, complete with all Kodo API extensions. Behind the scenes, however, the actions you take on a client persistence manager are sent to the corresponding server-side persistence manager for processing. For performance reasons and because your persistent objects are not proxies themselves, each client persistence manager has a local cache of managed objects, synchronized with the server-side persistence manager's cache.

You obtain client persistence managers in the same way you obtain local persistence managers: from a persistence manager factory that you have constructed through JCA or the `JDOHelper` or `KodoHelper`. Client configuration properties are the same as those used for local Kodo operation, with the following exceptions:

- The `javax.jdo.PersistenceManagerFactoryClass` property must be set to Kodo's client factory, `kodo.remote.ClientPersistenceManagerFactory`.
- The `kodo.PersistenceManagerServer` setting is used to find the remote server. If you are using a **standalone persistence manager server**, the value of this property is typically the same as its value on the server. If you are using the **HTTP servlet**, the value of this property on the client is:

```
kodo.PersistenceManagerServer: http(URL=<servlet-url>)
```

`http` in the setting above is an alias for the `com.solarmetric.remote.HTTPTransport`, whose `URL` property indicates the URL to connect to. You can also specify a `Decorators` property, as discussed in [Section 11.2.4, “Data Compression and Filtering”](#) [307]

- The `kodo.ConnectionRetainMode` property controls how the client handles connections to the server, not how the

server handles connections to the database. The available values are the same as the options for database connections:

- **persistence-manager**: Each client persistence manager obtains a single connection to its server-side counterpart and uses this connection until the persistence manager is closed.
- **transaction**: A connection is obtained when each transaction begins, and relinquished when the transaction completes. Nontransactional connections are obtained as needed and released immediately.
- **on-demand**: A connection to the server is obtained when needed, and immediately closed when the request has been fulfilled. This is the default.
- **kodo.ConnectionFactoryProperties** controls pooling not for database connections, but for connections from the client machine to the remote server. The following pooling options are available:
  - **ExceptionAction**: The action to take when a connection that has thrown an exception is returned to the pool. Set to **destroy** to destroy the connection. Set to **validate** to validate the connection (subject to the **TestOnReturn**, **TestOnBorrow**, and other test settings). Set to **none** to ignore the fact that the connection has thrown an exception, and assume it is still usable. Defaults to **destroy**.
  - **MaxActive**: The maximum number of connections in use at one time. Defaults to 8.
  - **MaxIdle**: The maximum number of idle connections to keep in the pool. Defaults to 8.
  - **MaxWait**: The maximum number of milliseconds to wait for a free connection to become available before giving up. Defaults to 3000.
  - **MinEvictableIdleTimeMillis**: The minimum number of milliseconds that a connection can sit idle before it becomes a candidate for eviction from the pool. Defaults to 30 minutes. Set to 0 to never evict a connection based on idle time alone.
  - **TestOnBorrow**: Whether to validate connections before obtaining them from the pool. Defaults to **true**.
  - **TestOnReturn**: Set to **true** to validate connections when they are returned to the pool.
  - **TestWhileIdle**: Set to **true** to periodically validate idle connections.
  - **TimeBetweenEvictionRunsMillis**: The number of milliseconds between runs of the eviction thread. Defaults to -1, meaning the eviction thread will never run.
  - **ValidationTimeout**: The minimum number of milliseconds that must elapse before a connection will ever be re-validated. This property is usually used with **TestOnBorrow** or **TestOnReturn** to reduce the number of validations performed, because the same connection is often borrowed and returned many times in short periods of time. Defaults to 300000 (5 minutes).
  - **WhenExhaustedAction**: The action to take when there are no available connections in the pool. Set to **exception** to immediately throw an exception. Set to **block** to block until a connection is available or the maximum wait time is exceeded. Set to **grow** to automatically grow the pool. Defaults to **block**.

Remember that persistent connections to the server consume server-side resources, and therefore should be minimized if possible. To disable pooling altogether, set **MaxActive** to 0.

- Database connectivity and JDBC-related properties are ignored by the client persistence manager factory. All database communication takes place on the server, so these properties are only valid on the server-side persistence manager factory. There are, however, two exceptions to this rule. If specified, the client will transfer your local **javax.jdo.option.ConnectionUserName** and **javax.jdo.option.ConnectionPassword** settings to the server. This allows different remote clients to connect as different database users.

Other than the bullet points above, you configure client persistence manager factories in the same way as local factories. Keep in mind, though, that the configuration you specify on the client only applies to the client factory, not the server. For example, if you configure a data cache and query cache on the client, these caches will only "see" changes made by the client; they will not automatically synchronize with changes made by any other client or changes made on the server. Thus, you will typically want to configure components like the data cache, query cache, lock manager, etc. on the server only (where clients can still benefit from them by proxy), and turn them off on the client.

### *Example 11.5. Client Configuration*

```
javax.jdo.PersistenceManagerFactoryClass: kodo.remote.ClientPersistenceManagerFactory
javax.jdo.option.Multithreaded: false
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.RetainValues: true

kodo.LicenseKey: xxxx
kodo.PersistenceManagerServer: tcp(Host=jdohost.mydomain.com, Port=5555)
kodo.ConnectionRetainMode: transaction
kodo.ConnectionFactoryProperties: MaxIdle=3, ValidationTimeout=60000
```

### *Example 11.6. HTTP Client Configuration*

```
javax.jdo.PersistenceManagerFactoryClass: kodo.remote.ClientPersistenceManagerFactory
javax.jdo.option.Multithreaded: false
javax.jdo.option.Optimistic: true
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.RetainValues: true

kodo.LicenseKey: xxxx
kodo.PersistenceManagerServer: http(URL=http://jdohost.mydomain.com/tomcat/pmserver)
kodo.ConnectionRetainMode: transaction
kodo.ConnectionFactoryProperties: MaxIdle=3, ValidationTimeout=60000
```

---

## 11.2.4. Data Compression and Filtering

Kodo's built in transport implementations -- `tcp`, `http` -- allow you to wrap their data streams in decorators to add additional functionality such as data compression and filtering. Each accepts a `Decorators` configuration property specifying a semi-colon-separated list of `com.solarmetric.remote.StreamDecorators` to decorate the input and output streams between the client and server. Each item in the list can be the full class name of a custom decorator, or one of the following built-in aliases:

- `gzip`: Use gzip compression when transferring data.

### *Example 11.7. Enabling Compression with the TCP Transport*

On the server:

```
kodo.PersistenceManagerServer: tcp(Port=5555, Decorators=gzip)
```

On the client:

```
kodo.PersistenceManagerServer: tcp(Host=jdohost.mydomain.com, Port=5555, Decorators=gzip)
```

### ***Example 11.8. Enabling Compression with the HTTP Transport***

In the server properties file / servlet configuration:

```
kodo.PersistenceManagerServer: http(Decorators=gzip)
```

On the client:

```
kodo.PersistenceManagerServer: http(URL=http://jdohost.mydomain.com/tomcat/pmserver, Decorators=gzip)
```

---

## **11.2.5. Remote Persistence Manager Deployment**

---

Using Kodo's remote features involves deploying Kodo to the server machine as well as one or more client machines. Deploying Kodo on the server is exactly the same as deploying Kodo for local use. You must include all Kodo libraries, your configuration properties file (if you use one), your logging configuration file (again, if you use one), your JDBC drivers, your enhanced persistent classes, your metadata, and your O/R mapping information. All of these topics are covered in other sections of this manual.

Deploying Kodo on the client is also the same as deploying Kodo for local use, with two small exceptions:

1. JDBC libraries are not required on the client.
2. O/R mapping information is not required on the client.

Note that you may include the above information in your deployment; it is simply not required.

---

# Chapter 12. Management and Monitoring

The management and monitoring capability uses the Java Management Extensions (JMX) standard to allow for both local and remote management of key:

- Attributes
- Operations
- Notifications / Performance Statistics

The unit of management in JMX is the Managed Bean (MBean). Kodo provides a number of MBeans that allow management of key components. For example, the connection pool MBean provides statistics on the numbers of active and idle connections, and allows modification of the attributes controlling the maximum number of active connections.

The default JMX implementation used by Kodo is MX4J version 1.1.1 (see [mx4j.sourceforge.net](http://mx4j.sourceforge.net)). The MX4J jar, `mx4j-jmx.jar`, is included in the Kodo distribution, and is covered under the MX4J license, which can be found in the `lib` directory. To use a different implementation, simply remove the `mx4j-jmx.jar` file from your `CLASSPATH` and insert the appropriate jars for your JMX implementation.

Note that the resource archive `kodo.rar` provided with Kodo does not include MX4J. This is because most application servers provide their own JMX implementation. However, a small subset of the MX4J implementation is included.

The management and monitoring capability can be run both locally and remotely. An example of its use can be found in the `management` sample described in [Section 1.13, “JMX Management” \[371\]](#)

## 12.1. Configuration

---

The Management capability is configured via the standard Kodo configuration system using the `kodo.ManagementConfiguration` property (see [Section 2.6.27, “kodo.ManagementConfiguration” \[151\]](#)). This property is also a plugin string (see [Section 2.4, “Plugin Configuration” \[140\]](#)), so you can also set it to the full class name of a custom `ManagementConfiguration`. Pre-defined values are:

- `none`: No management or profiling turned on. This is the default.
- `profiling-gui`: Turn on the local profiling GUI (see [Section 14.7.1, “Profiling in an embedded GUI” \[343\]](#) for more configuration information).
- `profiling-export`: Export profiling data (see [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#) for more configuration information).
- `profiling`: Enable profiling without export or GUI. Useful when trying to access the `ProfilingAgent` programmatically.
- `mgmt`: Enable management. Suitable for use in JBoss and other environments where all MBeans should be registered with a JMX MBeanServer for either management via a user defined mechanism, or via a mechanism defined by the MBeanServer. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[311\]](#)).
- `mgmt-prof`: Enable management, including profiling. Suitable for use in JBoss and other environments where all MBeans should be registered with a JMX MBeanServer for either management via a user defined mechanism, or via a mechanism defined by the MBeanServer. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[311\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#).

- `mgmt-export`: Enable management, and enable a profiling export. Suitable for use in JBoss and other environments where all MBeans should be registered with a JMX MBeanServer for either management via a user defined mechanism, or via a mechanism defined by the MBeanServer. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#) and [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#)
- `mx4j1-remote-mgmt`: Enable remote management via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[31\]](#))
- `mx4j1-remote-mgmt-prof`: Enable remote management, including remote profiling via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#)
- `mx4j1-remote-mgmt-export`: Enable remote management, and enable a profiling export via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#) and [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#)
- `jmx2-remote-mgmt`: Enable remote management via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[31\]](#))
- `jmx2-remote-mgmt-prof`: Enable remote management, including remote profiling via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#)
- `jmx2-remote-mgmt-export`: Enable remote management, and enable a profiling export via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#) and [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#)
- `local-mgmt`: Enable management, bringing up the JMX management console in the local JVM. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#))
- `local-mgmt-prof`: Enable management, including profiling, bringing up the JMX management console in the local JVM. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#)
- `local-mgmt-export`: Enable management, bringing up the JMX management console in the local JVM, and enable a profiling export. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#) and [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#)
- `wl81-mgmt`: Enable WebLogic 8.1 management. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[31\]](#)) and the `WebLogic 8.1 Group` (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[31\]](#))

- `wl81-mgmt-prof`: Enable WebLogic 8.1 management, including remote profiling. Supports optional parameters in the Management Group (see [Section 12.1.1, “Optional Parameters in Management Group” \[311\]](#)) and the WebLogic 8.1 Group (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[312\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#)
- `wl81-mgmt-export`: Enable WebLogic 8.1 management, and enable a profiling export. Supports optional parameters in the Management Group (see [Section 12.1.1, “Optional Parameters in Management Group” \[311\]](#)) and the WebLogic 8.1 Group (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[312\]](#)). Also supports optional parameters described in [Section 14.7.3, “Controlling how the profiler obtains context information” \[345\]](#) and [Section 14.7.2, “Dumping profiling data to disk from a batch process” \[345\]](#)

## 12.1.1. Optional Parameters in Management Group

---

Those `kodo.ManagementConfiguration` plugins that provide JMX based management have three optional parameters, described in this section.

If JMX based management is desired, Kodo needs to either find an existing `MBeanServer` or create a new one. Appropriate plugins for the configuration property `kodo.ManagementConfiguration` have an optional parameter `MBeanServerStrategy` that allows for configuration of this process. The following options are available:

- `any-create`: Attempt to find an existing `MBeanServer`. If multiple are found, use the first one. If none are found, create a new server. This is the default.
- `create`: Do not attempt to find an existing `MBeanServer`. Always create a new server.
- `agentId:<agent ID>`: Attempt to find an existing `MBeanServer` with the given agent ID. If such a server is not found, create a new server.

For example, in order to force creation of a new `MBeanServer` when doing remote management, include the following line in your properties:

- `kodo.ManagementConfiguration: remote-mgmt(MBeanServerStrategy="create")`

The other optional parameters are: `EnableLogMBean` and `EnableRuntimeMBean`.

- `EnableLogMBean`: When set to true, indicates that the `LogMBean` should be registered.
- `EnableRuntimeMBean`: When set to true, indicates that the `RuntimeMBean` should be registered.

For example, to register both the `RuntimeMBean` and the `LogMBean`:

- `kodo.ManagementConfiguration: local-mgmt  
(EnableLogMBean=true,EnableRuntimeMBean=true)`

## 12.1.2. Optional Parameters in Remote Group

---

In order to do remote management, a remote JMX adaptor needs to be started. Once the adaptor has started, the GUI needs to be started. See [Section 12.2, “Kodo Management Console” \[313\]](#) for more information. The Remote Group enables configuration of the remote management adaptor. For adaptors other than the `mx4j-jrmp` adaptor, a custom `ManagementConfiguration` must be defined. By default, the `mx4j-jrmp` adaptor sets up an RMI registry naming service on

`rmi://localhost:1099` and registers an MX4J JRMP JMX adaptor under the JNDI name `jrmmp`. To change the host, port and JNDI name used, the following optional parameters are available on those `kodo.ManagementConfiguration` plugins that support the `Remote Group`.

- `JNDIName`: The JNDI name under which to register the remote JMX adaptor. Defaults to `jrmmp`.
- `host`: The hostname on which the RMI registry naming service will listen. Defaults to `localhost`.
- `port`: The port on which the RMI registry naming service will listen. Defaults to 1099.

For example, to listen on port 2345:

```
kodo.ManagementConfiguration: remote-mgmt(port=2345)
```

### 12.1.3. Optional Parameters in JSR 160 Group

---

In order to do remote management using the JSR 160 standard (supported by JMX implementations supporting JMX 1.2 and higher), a remote JMX adaptor needs to be started. Once the adaptor has started, the GUI needs to be started. See [Section 12.2, “Kodo Management Console” \[313\]](#) for more information. The JSR 160 Group enables configuration of the remote management adaptor.

For convenience, the JSR 160 adaptor sets up an RMI registry naming service on `rmi://localhost:1099` and registers the JMX Connector Server with it, by default. To change the host, port and Service URL used, the following optional parameters are available on those `kodo.ManagementConfiguration` plugins that support the `JSR 160 Group`.

- `serviceURL`: The JMX Service URL name under which to register the JMX Connector Server. Defaults to `service:jmx:rmi://localhost/jndi/jmxservice`, indicating that the RMI connector will be used and registered under the JNDI name `jmxservice`.
- `namingImpl`: The classname of the naming service implementation to start in order to register the RMI connector with a JNDI name. Defaults to `mx4j.tools.naming.NamingService`, which is appropriate for MX4J v. 2.x. If set to the empty string, no naming service will be started. This is appropriate if a naming service is already running, or if a non-RMI connector is used.
- `host`: The hostname on which the RMI registry naming service will listen. Defaults to `localhost`. This parameter is ignored for connectors that do not register with a naming service.
- `port`: The port on which the RMI registry naming service will listen. Defaults to 1099. This parameter is ignored for connectors that do not register with a naming service.

For example, to have the RMI registry naming service listen on port 2345:

```
kodo.ManagementConfiguration: jmx2-remote-mgmt(port=2345)
```

### 12.1.4. Optional Parameters in WebLogic 8.1 Group

---

The following parameters must be set on those `kodo.ManagementConfiguration` plugins that support the `WebLogic 8.1 Group`. Once WebLogic has started, the GUI needs to be started. See [Section 12.2, “Kodo Management Console” \[313\]](#)

for more information. For additional requirements in order to do remote management, please see [Section 12.2.1.1, “Connecting to Kodo under WebLogic 8.1” \[314\]](#)

- `username`: The username that Kodo should use to access the WebLogic MBeanServer. This must be set.
- `password`: The password that Kodo should use to access the WebLogic MBeanServer. This must be set.
- `serverName`: The server name of the server whose MBeanServer to which Kodo should connect. This must be set.
- `url`: The URL to which Kodo should connect to access the WebLogic MBeanServer. Defaults to `t3://localhost:7001`.

For example:

```
kodo.ManagementConfiguration: wl81-mgmt(username="admin",password="admin",serverName="myserver",url="t3://localhost:7001")
```

---

## 12.1.5. Configuring Logging for Management / Monitoring

Logging for the management and monitoring API is on the `com.solarmetric.Manage` logging channel.

---

## 12.2. Kodo Management Console

The Kodo Management Console is used for local and remote management of MBeans. It can be used to connect to a local MBean server or multiple remote MBeanServers. To connect to a local server, see [Section 12.1, “Configuration” \[309\]](#).

---

### 12.2.1. Remote Connection

To start the Kodo Management Console for remote management, run the `remotejmxtool` command. The `remotejmxtool` accepts the following arguments:

- `-connect/-c`: Whether to attempt an initial connection to the remote JMX adaptor. Defaults to `false`.
- `-type/-t`: The type of the remote JMX adaptor. Current supported types are `mx4j1`, `jmx2`, `weblogic81` and `jboss`. Defaults to `mx4j1`. Integration with other JMX server implementations that support remote connectivity can be accomplished by creating a class that implements the `RemoteMBeanServerFactory` interface. In this case, the type should be the fully qualified name of the implementing class.
- `-host/-h`: Hostname of the JNDI service provider where the remote JMX adaptor is registered. Defaults to `localhost`. When attempting an initial connection to WebLogic, this must be set to a hostname of the form `user-name:password@hostname`. This is optional for JSR 160 connectors, as it may not be necessary for some connectors, and may be encoded in the JMX Service URL for others.
- `-port/-p`: Port of the JNDI service provider where the remote JMX adaptor is registered. Defaults to 1099 when connecting to MX4J. Defaults to 7001 when connecting to WebLogic. This is optional for JSR 160 connectors, as it may not be necessary for some connectors, and may be encoded in the JMX Service URL for others.
- `-name/-n`: For non-JSR 160 connectors, the JNDI name of the remote JMX adaptor. Defaults to a special value `default` which yields the default JNDI name appropriate for the chosen remote JMX adaptor type. For MX4J, the default is `jrmf`, and for JBoss, the default is the first available JMX adaptor at the specified JNDI service provider. For WebLogic, this parameter is ignored. For JSR 160 connectors, this is the JMX Service URL, and defaults to `service:jmx:rmi://localhost/jndi/jmxservice`. Note that this can also encode the host and port parameters, if desired. For example, the default JMX Connector Server could be referenced by `ser-`

vice:jmx:rmi:///localhost/jndi/rmi:///localhost:1099/jmxservice. In that case, the host and port parameters will be ignored.

For example, to automatically connect to the MX4J remote JMX adaptor on host `myhost.mydomain.com`, use the following command:

```
remotejmxtool -c -host myhost.mydomain.com
```

Once `remotejmxtool` is up, you can connect to multiple remote JMX adaptors. To connect to Kodo with MX4J v. 1.1.x, select **Connect to Kodo JMX...** from the File menu. To connect to Kodo with a JSR 160 connector, select **Connect to Kodo JMX 1.2...** from the File menu. To connect to Kodo running under WebLogic, select **Connect to Kodo via WebLogic JMX...** from the File menu. To connect to Kodo running under JBoss, select **Connect to Kodo via JBossMX...** from the File menu.

### 12.2.1.1. Connecting to Kodo under WebLogic 8.1

---

In order to connect to WebLogic 8.1 with `remotejmxtool`, the following requirements must be met:

- `remotejmxtool` must be run with the `weblogic.jar` (found in the `weblogic81/server/lib/` directory of the WebLogic 8.1 distribution) in your CLASSPATH. Note that this library should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your CLASSPATH.
- The `remotejmxtool` must be run with JDK 1.4.x.
- The jar `kodo-wl81manage.jar` must be put in the WebLogic system CLASSPATH. You can accomplish this by editing `startWebLogic.sh/.cmd`.

### 12.2.1.2. Connecting to Kodo under JBoss 3.2

---

In order to connect to JBossMX 3.2, `remotejmxtool` must be run with the following libraries from the JBoss distribution in your CLASSPATH.

- `jboss-common-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.
- `jboss-jmx.jar`: Found in the `lib/` directory of the JBoss 3.2 distribution.
- `jmx-adaptor-plugin.jar`: Found in the `server/all/lib/` directory of the JBoss 3.2 distribution.
- `jnp-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.
- `jboss-system.jar`: Found in the `lib/` directory of the JBoss 3.2 distribution.
- `jnet.jar`: Found in the `client/` directory of the JBoss 3.2 distribution. Alternately, `remotejmxtool` can be run under JDK 1.4 or higher.
- `concurrent.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.
- `jbossall-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.

Note that these libraries should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your CLASSPATH.

### 12.2.1.3. Connecting to Kodo under JBoss 4

---

In order to connect to JBossMX 4, `remotejmxtool` must be run with the following libraries from the JBoss distribution in your CLASSPATH.

- `jboss-common-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jboss-jmx.jar`: Found in the `lib/` directory of the JBoss 4 distribution.
- `jmx-adaptor-plugin.jar`: Found in the `server/all/lib/` directory of the JBoss 4 distribution.
- `jnp-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jboss-system.jar`: Found in the `lib/` directory of the JBoss 4 distribution.
- `concurrent.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jbossall-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `dom4j.jar`: Found in the `lib/` directory of the JBoss 4 distribution.

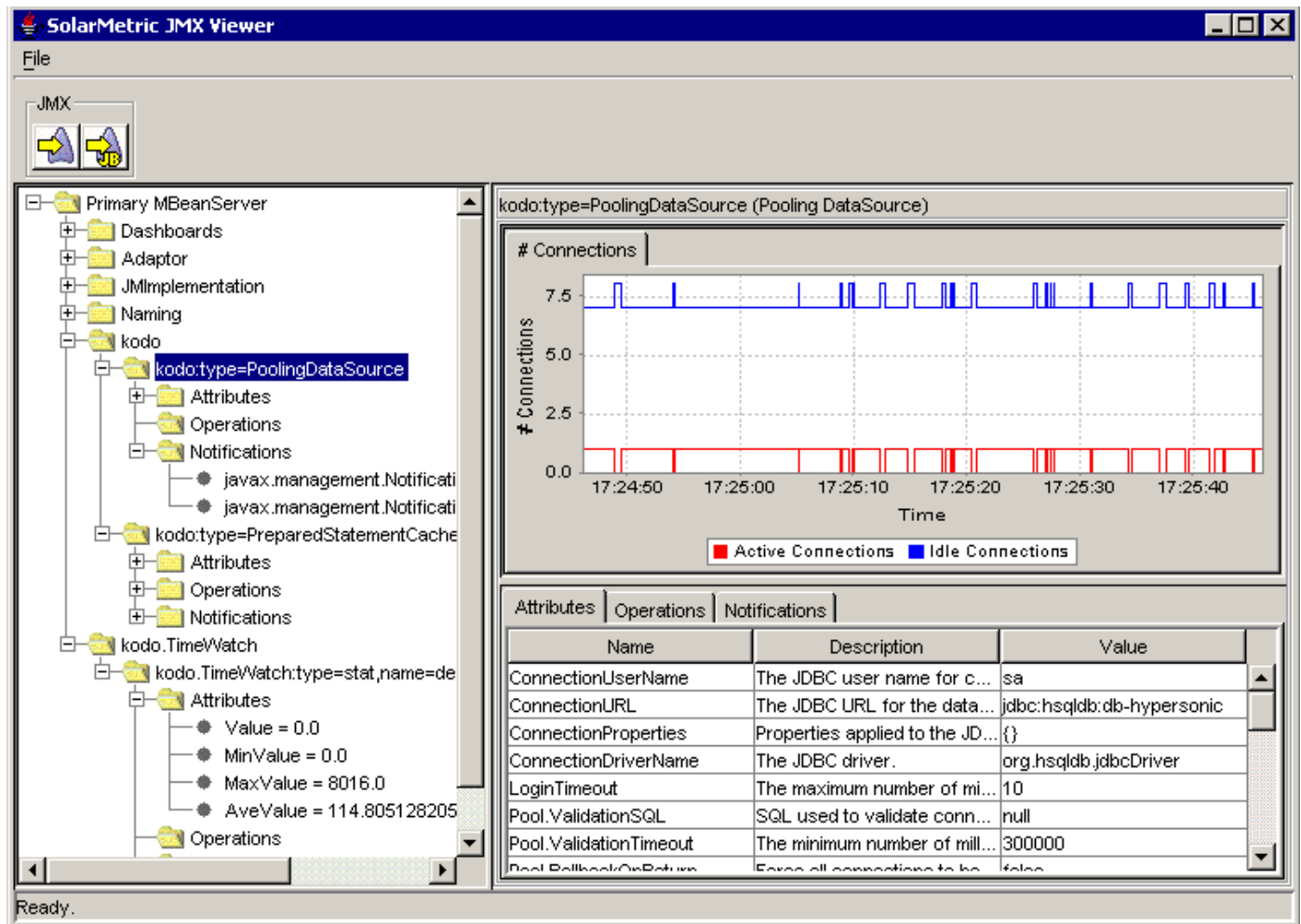
Note that these libraries should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your CLASSPATH.

Additionally, the following requirements must be met:

- The `remotejmxtool` must be run with JDK 1.5.x.
- The jar `kodo-jboss4manage.jar` must be put in the JBoss 4 system CLASSPATH. You can accomplish this by placing the jar in the server's `lib/` directory (e.g. `>JBoss 4 install</server/default/lib/`).

## 12.2.2. Using the Kodo Management Console

---



The above diagram shows the Kodo Management Console window. The Kodo Management Console window is divided into two main parts, the JMX Explorer on the left, and the MBean Panel on the right.

### 12.2.2.1. JMX Explorer

The JMX Explorer provides a tree view of the connected MBeanServers. Under each MBeanServer are the JMX domains handled by that server. Under each domain are the MBeans within that domain. Under each MBean are the Attributes, Operations and Notifications provided by that MBean.

#### 12.2.2.1.1. Executing Operations

In order to execute an Operation of an MBean, right click on the Operation, and select "Execute..." from the context menu. A dialog box will come up asking for values for each of the arguments to the managed Operation. Fill in each of the values and hit the OK button to execute the Operation.

#### Note

Currently, only primitive types, primitive wrapper types and classes for which there is a constructor that takes a single String argument can be entered.

If the Operation returns a non-null value, the string representation of the return value will be shown.

### 12.2.2.1.2. Listening to Notifications

---

When an MBean is selected in the `JMX Explorer`, the Kodo Management Console automatically listens to all Notifications. To stop listening to all Notifications for a given MBean, right click on the `Notifications` node and select `Stop Listening All`. To stop listening to a single Notification, right click on the individual Notification and select `Stop Listening`. In order to listen to all Notifications provided by an MBean, right click on the `Notifications` node under the MBean and select `Listen All`. To listen to a single Notification, right click on the individual Notification and select `Listen`.

The available Notifications can be seen in the `MBean Panel` to the right of the `JMX Explorer`.

### 12.2.2.2. MBean Panel

---

The `Attributes`, `Operations` and `Notifications` of an MBean can be viewed in the `MBean Panel`. The top half of the panel shows Notifications and statistics, while the bottom half allows for viewing / editing `Attributes`, viewing available `Operations`, and viewing available Notifications.

#### 12.2.2.2.1. Notifications / Statistics

---

The top half of the `MBean Panel` shows the Notifications emitted by the selected MBean. Note that Notifications must be listened to (see [Section 12.2.2.1.2, “Listening to Notifications” \[317\]](#)) in order to be viewed in the `MBean Panel`. There is one tab per Notification. Certain Notifications represent statistics. These Notifications are grouped under tabs based on their ordinate description. Statistic Notifications are represented in charts. Dragging a rectangle across a chart causes the chart to zoom in on the selected area. Right clicking on a chart brings up a context menu with a number of options:

- `Properties...:` Edit chart properties, such as colors and labels.
- `Save as...:` Save the chart to disk.
- `Print...:` Print the chart.
- `Zoom In / Zoom Out:` Zoom in and out on either or both axes.
- `Auto Range:` Set the either or both the abscissa and ordinate range to see all of the values.

#### 12.2.2.2.2. Setting Attributes

---

The `Attributes` tab in the bottom half of the `MBean Panel` allows for viewing / editing of `Attributes`. Not all `Attributes` are editable. Selecting an editable `Attribute` allows you to set the value.

#### Note

Currently, only primitive types, primitive wrapper types and classes for which there is a constructor that takes a single `String` argument can be entered.

## 12.3. Accessing the MBeanServer from Code

---

You can access the `MBeanServer` in which the Kodo MBeans are registered using the `JDOConfiguration` interface's `getMBeanServer ()` method.

### *Example 12.1. Accessing the MBeanServer*

```
KodoPersistenceManagerFactory kpmf = ...;
MBeanServer mbServer = kpmf.getConfiguration ().getMBeanServer ();
```

## 12.4. MBeans

---

### 12.4.1. Log MBean

---

The Log MBean allows for remote monitoring of Log messages (see [Chapter 3, \*Logging\* \[161\]](#)). The MBean has a single Notification that, if listened to, will add an appender to the root logger that will send log messages as notifications. This MBean currently only provides log messages when using the Log4J logging service. The name of the Log MBean is `kodo:type=log,name=LogMBean`.

### 12.4.2. Kodo Pooling DataSource MBean

---

The Kodo Pooling DataSource (see [Section 4.1, “Using the Kodo JDO DataSource” \[166\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=PoolingDataSource`.

The Kodo DataSource has a number of Attributes which allow for viewing / editing of pool settings, and viewing of pool statistics. The Kodo Pooling DataSource also has a number of Statistic Notifications.

### 12.4.3. PreparedStatement Cache MBean

---

The PreparedStatement Cache (see `MaxCachedStatements` under [Section 4.1, “Using the Kodo JDO DataSource” \[166\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=PreparedStatementCache`.

The cache has a single `MaxCachedStatements` Attribute for viewing / editing the number of statements to cache, and a number of Attributes for viewing cache statistics. The cache also has a number of Statistic Notifications.

### 12.4.4. Query Cache MBean

---

The Query Cache (see [Section 14.3.3, “Query Caching” \[332\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=QueryCacheImpl`.

The cache has a `CacheSize` Attribute for viewing / editing the size of the LRU cache, and a `SoftReferenceSize` Attribute for editing the size of the soft cache. It also has a number of Attributes for viewing cache statistics. The cache also has a `clear` operation and a number of Statistic Notifications.

### 12.4.5. Datastore Cache MBean

---

Each Datastore Cache (see [Section 14.3, “Datastore Cache” \[329\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=DataCacheImpl,name=<cache name>` where `<cache name>` is the name of the cache as defined in `data-cache` metadata extension (see [Section 6.2.4.7, “data-cache” \[206\]](#)).

Each cache has a number of Attributes which allow for viewing / editing of cache settings, and viewing of cache statistics. Each cache also has a number of Statistic Notifications.

### 12.4.6. TimeWatch MBean

---

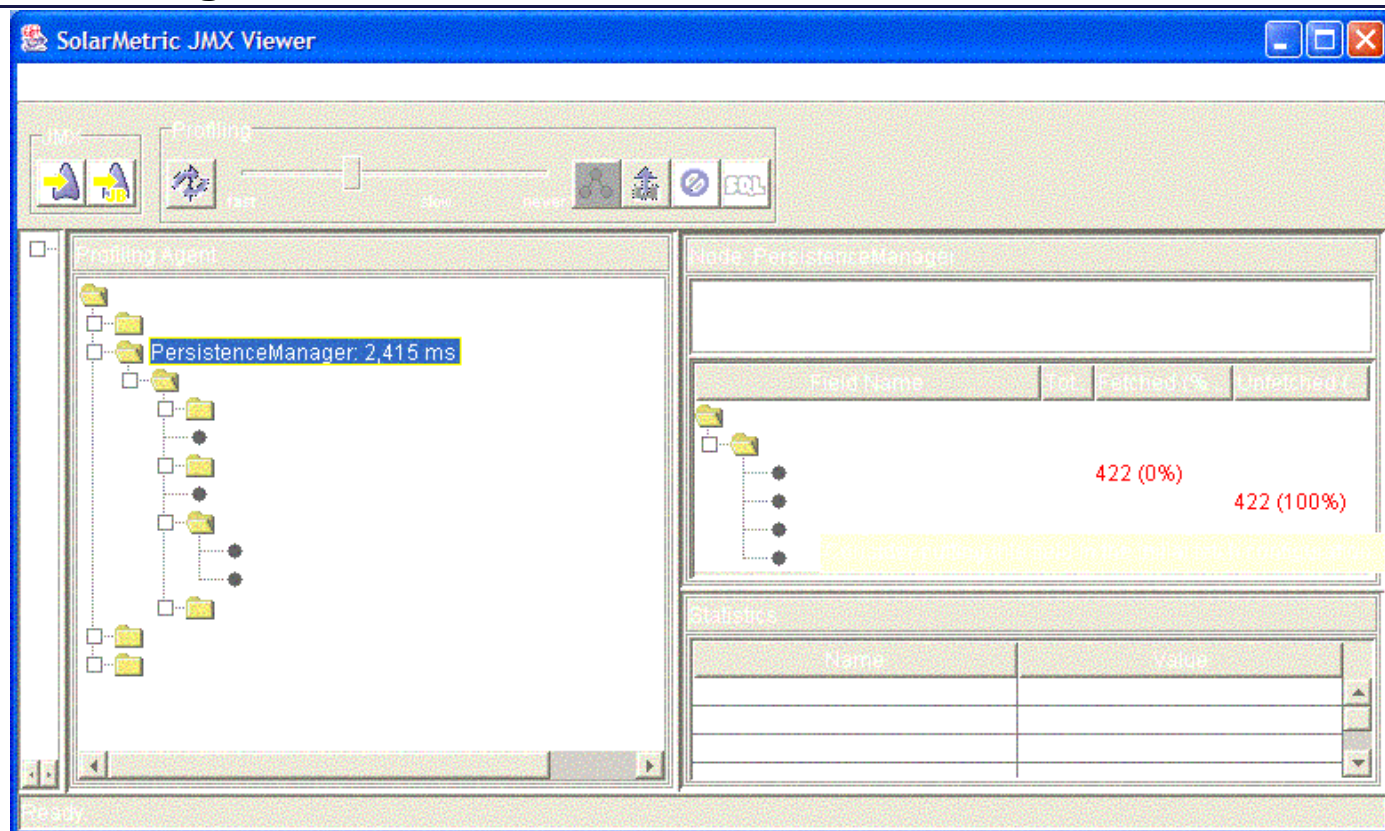
Each TimeWatch Statistic is managed by an MBean. For information on creating TimeWatch Statistics, see [TimeWatch](#) and [KodoTimeWatchManager](#) in the Kodo Javadoc. The name of the MBean is `kodo.TimeWatch:type=stat,name=<watchable name>,stat=<block name>` where `<watchable name>`

is the name of the TimeWatch and where `<block name>` is the name of the named code block.

## 12.4.7. Runtime MBean

The Runtime MBean allows for remote monitoring of the JVM Runtime in which Kodo is running. The MBean provides attributes for viewing the amount free memory available, and the total memory allocated to the JVM. It also has Statistic Notifications for free and total memory. The name of the Log MBean is `kodo:type=runtime`.

## 12.4.8. Profiling MBean



The Kodo Profiling MBean allows for profiling of application code. It is designed to help optimize the use of Kodo, and is not intended to be a generic profiling tool. Only Kodo specific APIs are instrumented. This section describes how to setup the profiling capability within the Kodo Management Console. For more information about the profiling capability, see [Section 14.7, “Profiling” \[342\]](#)

To use the Profiling MBean, perform the following steps:

- Enable the MBean: Set the `kodo.ManagementConfiguration` property to `remote-mgmt-prof` or `local-mgmt-prof`.
- Start the Kodo Management Console either locally (see [Section 12.1, “Configuration” \[309\]](#)) or remotely (see [Section 12.2.1, “Remote Connection” \[313\]](#)).
- Select the MBean. The name of the MBean is `kodo:type=Profiling`.
- Listen to the MBean Notifications. See [Section 12.2.2.1.2, “Listening to Notifications” \[317\]](#) (this happens automatically, but if you turn off notifications, you will need to turn them on again in order to see updates).

The MBean Panel contains a custom viewer when the Profiling MBean is selected. This custom viewer contains the Kodo Profiling Console. Additionally, the console toolbar will have profiling toolbar options. Please see **Section 14.7, “Profiling” [342]** for more information.

---

# Chapter 13. Enterprise Edition

The features we've discussed thus far are all included in the Kodo JDO Standard Edition. This chapter presents features unique to the Kodo JDO Enterprise Edition. The Enterprise Edition primarily differs from the Standard Edition in its ability to integrate with application servers' global transactions and its advanced custom query capabilities. The Enterprise Edition also includes all the components of the **Performance Pack**, which we describe in the next chapter.

## 13.1. Integrating with the Transaction Manager

---

Kodo JDO persistence managers have the ability to automatically synchronize their transactions with an external transaction manager. Whether or not persistence managers from a given persistence manager factory exhibit this behavior by default depends on the factory's **kodo.TransactionMode** configuration property. The property can take the following values:

- **local**: Perform transaction operations locally.
- **managed**: Integrate with the application server's managed global transactions.

You can override the global transaction mode setting when you obtain a persistence manager by using the **KodoPersistenceManagerFactory**'s `getPersistenceManager(boolean managed, int connRetainMode)` method.

In order to use global transactions, Kodo JDO must be able to access the application server's `javax.transaction.TransactionManager`. Kodo JDO can automatically discover the transaction manager for most major application servers. Occasionally, however, you might have to point Kodo JDO to the transaction manager for an unrecognized or non-standard application server setup. This is accomplished through the **kodo.ManagedRuntime** configuration property. This property describes a **kodo.ee.ManagedRuntime** implementation to use for transaction manager discovery. You can specify your own implementation, or use one of the built-ins:

- **auto**: This is the default. It is an alias for the **kodo.eeAutomaticManagedRuntime** class. This managed runtime is able to automatically integrate with several common application servers.
- **invocation**: An alias for the **kodo.ee.InvocationManagedRuntime** class. You can configure this runtime to invoke any static method in order to obtain the appserver's transaction manager.
- **jndi**: An alias for the **kodo.ee.JNDIManagedRuntime** class. You can configure this runtime to look up the transaction manager at any JNDI location.

See the Javadoc for each class for details on the bean properties you can pass to these plugins in your configuration string.

### *Example 13.1. Configuring Transaction Manager Integration*

```
kodo.TransactionMode: managed
kodo.ManagedRuntime: jndi(TransactionManagerName=java:/TransactionManager)
```

## 13.2. XA Transactions

---

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.

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.

### 13.2.1. Requirements for Using Kodo with XA Transactions

---

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

- A managed environment that provides an XA compliant transaction manager. 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.

### 13.2.2. 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 Kodo to use your third-party `javax.sql.XADataSource` See [Section 4.2, “Using a Third-Party DataSource” \[168\]](#) on using a third-party data source for details.
- Configure a separate data source for non-transactional connections. This should *not* be an XA data source. Kodo needs to have access to a data source that will not be enlisted in an XA transaction for things like obtaining database sequence numbers for datastore identity, which should not be part of Kodo's transaction.

See [Section 4.1, “Using the Kodo JDO DataSource” \[166\]](#) or [Section 4.2, “Using a Third-Party DataSource” \[168\]](#) for information on how to configure the non-transactional data source. Just remember to use the "2" version of all the connection configuration properties: `kodo.Connection2DriverName`, `kodo.Connection2UserName`, `javax.jdo.option.ConnectionFactory2Name`, and so forth.

- Kodo needs to be aware that the XA data source is automatically enlisted in the distributed transaction. Set the `kodo.jdbc.DataSourceMode` configuration property to `enlisted`.

#### *Example 13.2. XA Configuration*

```
javax.jdo.option.ConnectionFactoryName: java:/OracleXADataSource
kodo.Connection2UserName: scott
kodo.Connection2Password: tiger
kodo.Connection2URL: jdbc:oracle:thin:@CROM:1521:JDODB
kodo.Connection2DriverName: oracle.jdbc.driver.OracleDriver
kodo.ConnectionFactory2Properties: MaxActive=20, MaxIdle=10
kodo.TransactionMode: managed
```

```
kodo.jdbc.DataSourceMode: enlisted
```

## 13.3. JDOQL Subqueries

The JDO specification does not provide a simple way to embed one query within another. The best you can do is perform the "inner" query, and use its result(s) as a parameter to the "outer" query. Kodo JDO Enterprise Edition corrects this oversight with full support for JDOQL subqueries.

Kodo's JDOQL subqueries utilize JDO 2's single-string JDOQL syntax, explained in [Section 11.9, "Single-String JDOQL"](#) [70]. Kodo makes one minor addition to the single-string format, requiring a logical alias for the subquery candidates. As you'll see below, this allows you to create *correlated* subqueries that include values from the parent query. The results of a subquery are supplied to the parent query as either a single value or a collection, depending on usage. Let's examine some examples of subqueries in action.

### Example 13.3. Comparison to Subquery

In this example, we find all magazines that are tied for the highest cover price.

```
Query q = pm.newQuery (Magazine.class, "price == (select max(m.price) "
+ "from org.mag.Magazine m)");
Collection mags = (Collection) q.execute ();
```

Notice that the example surrounds its subquery in parentheses. This is required for all subqueries. Also, note the use of the *m* alias for subquery candidates. The candidate alias is also required. Without it, Kodo couldn't support correlated subqueries, like the one in the next example.

### Example 13.4. Correlated Subquery

In this example, we find all the magazines tied for the highest price within their publisher. The subquery is *correlated* because it uses the *publisher* value of the parent query's candidate instance.

```
Query q = pm.newQuery (Magazine.class, "price == (select max(m.price) "
+ "from org.mag.Magazine m where m.publisher == publisher)");
Collection mags = (Collection) q.execute ();
```

The previous example used subqueries that were guaranteed to return exactly one result. What if the subquery might return no results, or multiple results? In these cases, you should treat the subquery as a collection. Use *contains* to test whether a value is included in the subquery results, and *isEmpty* to test whether the subquery has no results.

### Example 13.5. Subquery Contains

Find all magazines whose title matches the title of an article.

```
Query q = pm.newQuery (Magazine.class,
    "(select a.title from org.mag.Article a).contains (title)");
```

Filtering on whether a subquery contains a value is the same as using the IN operator in SQL.

### ***Example 13.6. Subquery Empty***

Find all magazines whose title does not match the title of an article.

```
Query q = pm.newQuery (Magazine.class,
    "(select from org.mag.Article a where a.title == title).isEmpty ()");
```

Testing whether a subquery is empty is equivalent to SQL's NOT EXISTS assertion. To perform an EXISTS test instead, just negate the expression: `!(<subquery>).isEmpty ()`

## **13.3.1. Subquery Parameters, Variables, and Imports**

---

Subqueries cannot declare their own parameters, variables, or imports. The outermost query must always specify all declarations for any subqueries it has, including nested subqueries. Subqueries can, however, introduce new implicit variables and parameters just by using them. For a refresher on JDOQL declarations, including implicit parameters and variables, see [Section 11.3, “Advanced Object Filtering” \[58\]](#).

## **13.4. Direct SQL Execution**

---

Kodo JDO Enterprise Edition allows you to execute SQL selects and stored procedures directly through the JDO query interface, retrieving matching objects or column projections rather than a low-level `ResultSet`. SQL queries can only be executed in the database; if you attempt to execute a SQL query under conditions that require in-memory evaluation, Kodo will throw an error.

SQL queries are documented in detail the JDO Overview's [Chapter 12, SQL Queries \[76\]](#). Also, the `StoredProcMain` driver program in the `samples/ormapping` directory of your Kodo distribution demonstrates a working SQL query.

## **13.5. MethodQL**

---

If JDOQL and direct SQL execution do not match your needs, Kodo JDO Enterprise Edition also allows you to name a Java method to use to load a set of objects. Kodo uses JDO's built-in alternate query language capabilities to create method-based queries. In a MethodQL query, the filter string names a static method to invoke to determine the matching objects:

```
// the method query language is 'kodo.MethodQL'
Query q = pm.newQuery ("kodo.MethodQL", null);

// set the type of objects that the method returns
q.setClass (Person.class);

// set the filter to the method to execute, including full class name; if
// the class is in the candidate class' package or in the query imports, you
// can omit the package; if the method is in the candidate class, you can omit
// the class name and just specify the method name
q.setFilter ("com.xyz.Finder.getByName");

// parameters are declared and passed the same way as in standard queries
q.declareParameters ("String firstName, String lastName");
```

```
// this executes your method to get the results
Collection results = (Collection) q.execute ("Fred", "Lucas");
```

For datastore queries, the method must have the following signature:

```
public static ResultObjectProvider xxx(KodoPersistenceManager pm,
    ClassMetaData meta, boolean subclasses, Map params, FetchConfiguration fetch)
```

The returned result object provider should produce objects of the candidate class that match the method's search criteria. If the returned objects do not have all fields in the given fetch configuration loaded, Kodo will make additional trips to the datastore as necessary to fill in the data for the missing fields.

In-memory execution is slightly different, taking in one object at a time and returning a boolean on whether the object matches the query:

```
public static boolean xxx(KodoPersistenceManager pm, ClassMetaData meta,
    boolean subclasses, Object obj, Map params, FetchConfiguration fetch)
```

In both method versions, the given params map contains the names and values of all the parameters declared in the JDO query object and passed to the execute method.

The `StoredProcQueries` class and `StoredProcMain` driver program in the `samples/ormapping` directory of your Kodo distribution demonstrate how you might implement your own custom query method, and how to execute it through the JDO query interface.

## 13.6. Remote PersistenceManagers

---

Kodo JDO Enterprise Edition includes support for multi-tier hardware configurations, in which a `PersistenceManager` in the client tier communicates with a middle-tier Kodo server machine which in turn communicates with the database. This architecture can be useful for highly-scalable systems with a large number of client machines, or when the client machines do not have direct access to the database. See [Section 11.2, “Remote Persistence Managers” \[303\]](#) for details about remote persistence managers.

## 13.7. Custom Class Mappings

---

Special database schemas or rules sometimes interfere with the standard ways that Kodo interacts with the database to create, delete, update, or load objects. In these situations, Kodo's custom class mapping capabilities can be critical. For example, you may have a requirement that all deletes happen via a stored procedure that does some auditing and user validation, or that loaded data should be filtered based on some application-specific authentication rules. These types of class-level data access requirements can be implemented by creating a custom class mapping. See [Section 7.6.5, “Custom Class Mapping” \[228\]](#) for details about custom class mappings.

## 13.8. Non-relational Database Access

---

It is possible to adapt Kodo to access a non-relational data store by creating an implementation of the `kodo.runtime.StoreManager` interface. Kodo JDO Enterprise Edition provides an abstract `StoreManager` implementation to facilitate this process. See the `kodo.abstractstore javadocs` for details. Additionally, see [Section 1.14, “XML Store Manager” \[372\]](#) for an example of how to extend this abstract store manager.

---

# Chapter 14. Performance Pack

The Kodo JDO Performance Pack is a suite of features that enhance the speed and functionality of the Kodo runtime. The Performance Pack is included in all Enterprise Edition licenses, or can be purchased separately. Contact [sales@solarmetric.com](mailto:sales@solarmetric.com) for details.

## 14.1. SQL Batching

---

In addition to connection pooling and prepared statement caching, Kodo can be configured to employ SQL batching to speed up JDBC updates. By default, SQL batching is enabled for any JDBC driver that supports it. When batching is on, Kodo JDO automatically orders its SQL statements to maximize the size of each batch. This can result in large performance gains for transactions that modify a lot of data.

SQL batching is configured through the system **DBDictionary**, which is controlled by the `kodo.jdbc.DBDictionary` configuration property. The example below shows how to enable and disable SQL batching via the configuration properties file.

### *Example 14.1. Configuring SQL Batching*

The batch limit is the maximum number of statements Kodo will ever batch together. A value of -1 means "no limit" (this is the default for most dictionaries). A value of 0 disables batching.

```
kodo.jdbc.DBDictionary: BatchLimit=25
```

## 14.2. Eager Fetching

---

Eager fetching is the ability to efficiently load subclass data and related objects along with the base instances being queried. Typically, Kodo JDO has to make a trip to the database whenever a relation is loaded, or when you first access data that is mapped to a table other than the least-derived superclass table. If you perform a query that returns 100 `Person` objects, and then you have to retrieve the `Address` for each person, Kodo may make as many as 101 queries (the initial query, plus one for the address of each person returned). Or if some of the `Person` instances turn out to be `Employees`, where `Employee` has additional data in its own table, Kodo once again might need to make extra database trips to access the additional employee data. With eager fetching, Kodo can reduce these cases to a single query.

Eager fetching only affects relations in the **fetch groups** being loaded. In other words, relations that would not normally be loaded immediately when retrieving an object or accessing a field are not affected by eager fetching. In our example above, the address of each person would only be eagerly fetched if the query were configured to include the address field or its fetch group, or if the address were in the default fetch group. This allows you to control exactly which fields are eagerly fetched in different situations. Similarly, JDO queries that exclude subclasses from the candidate extent aren't affected by eager subclass fetching, described below.

Eager fetching has three modes:

- **none**: No eager fetching is performed. Related objects are always loaded in an independent select statement, and no subclass data is loaded unless it is in the table(s) for the base type being queried.
- **join**: In this mode, Kodo joins to 1-1 relations in the configured fetch groups. If Kodo is loading data for a single instance, then by default Kodo will also join to any collection field in the configured fetch groups. When loading data for multiple instances, such as a query or extent, Kodo prefers to use *parallel* eager fetching for collections, as described below. You can

override these defaults for collection eager fetching using the metadata extension described in [Section 6.2.4.11](#), “**eager-fetch-mode**” [207].

Under `join` mode, Kodo uses a left outer join (or inner join, if the relations' field metadata sets the `null-value` attribute to `exception`) to select the related data along with the data for the target objects. This process works recursively for to-one joins, so that if `Person` has an `Address`, and `Address` has a `TelephoneNumber`, and the fetch groups are configured correctly, Kodo might issue a single select that joins across the tables for all three classes. To-many joins can not recursively spawn other to-many joins, but they can spawn recursive to-one joins.

Under the `join` subclass fetch mode, subclass data in other tables is selected by outer joining to all possible subclass tables of the type being queried. As you'll see below, subclass joining is configured separately from relation joining, and can be disabled for specific classes.

### Note

Some databases may not support outer joins. Also, Kodo JDO can not use left outer joins if you have set the `kodo.jdbc.DBDictionary` configuration parameter's `JoinSyntax` to property `traditional`.

- **parallel**: Under this mode, Kodo selects 1-1 relations and joined collections as described in the `join` mode description above. Unjoined collection fields, however, are eagerly fetched using a separate select statement for each collection, executed in parallel with the select statement for the target objects. The parallel selects use the where conditions from the primary select, but add their own joins to reach the related data. Thus, if you perform a query that returns 100 `Company` objects, where each company has a list of `Employee` objects and `Department` objects, Kodo will make 3 queries. The first will select the company objects, the second will select the employee objects for those companies, and the third will select the department objects for the same companies. Just as for joins, this process can be recursively applied to the objects in the relations being eagerly fetched. Continuing our example, if the `Employee` class had a list of `Projects` in one of the fetch groups being loaded, Kodo would execute a single additional select in parallel to load the projects of all employees of the matching companies.

Using an additional select to load each collection avoids transferring more data than necessary from the database to the application. If eager joins were used instead of parallel select statements, each collection added to the configured fetch groups would cause the amount of data being transferred to rise dangerously, to the point that you could easily overwhelm the network.

Parallel mode only applies to subclass data fetching when executing a JDO Query. In other situations, parallel mode acts just like `join` mode in regards to subclasses mapped to other tables. In the context of a Query, parallel mode instructs Kodo to execute one select per possible subclass, where each select will fetch all necessary data for instances of its corresponding class.

When Kodo knows that it is selecting for a single object only, such as in calls to `getObjectById`, it never uses `parallel` mode, because the additional selects can be made lazily just as efficiently. This mode only increases efficiency over `join` mode when multiple objects with eager relations are being loaded, or when multiple selects might be faster than joining to all possible subclasses.

## 14.2.1. Configuring Eager Fetching

You can control Kodo's default eager fetch mode through the `kodo.EagerFetchMode` and `kodo.SubclassFetchMode` configuration properties. Set each of these properties to one of the mode names described in the previous section: `none`, `join`, `parallel`. If left unset, the eager fetch mode defaults to `parallel` and the subclass fetch mode defaults to `join` (assuming you have purchased the performance pack).

You can also override the default fetch modes on individual Kodo persistence managers, queries, and extents. All of these components give you access to their internal `FetchConfiguration` object for controlling object loading behavior. The [runtime interfaces](#) chapter of this manual details these interfaces, including the `FetchConfiguration`.

### *Example 14.2. Setting the Default Eager Fetch Mode*

```
kodo.EagerFetchMode: join
kodo.SubclassFetchMode: parallel
```

### *Example 14.3. Setting the Eager Fetch Mode at Runtime*

```
import kodo.query.*;
import kodo.runtime.*;

...

KodoQuery kq = (KodoQuery) pm.newQuery (Person.class, "address.state == 'TX'");
FetchConfiguration fetch = kq.getFetchConfiguration ();
fetch.setEagerFetchMode (fetch.EAGER_JOIN);
fetch.setSubclassFetchMode (fetch.EAGER_PARALLEL);
Collection results = (Collection) kq.execute ();
```

You can specify a default subclass fetch mode for an individual class with the `subclass-fetch-mode` metadata extension, described in [Section 6.2.4.10, “subclass-fetch-mode” \[207\]](#). Note, however, that you cannot “upgrade” the runtime fetch mode with your class setting. If the runtime fetch mode is `none`, no eager subclass data fetching will take place, regardless of your metadata setting.

This applies to the `eager-fetch-mode` metadata extension as well (see [Section 6.2.4.11, “eager-fetch-mode” \[207\]](#)). You can use this extension to disable eager fetching on a field or to declare that a collection would rather use joins than parallel selects or vice versa. But an extension value of `join` won't cause any eager joining if the fetch configuration's setting is `none`.

---

## 14.2.2. Eager Fetching Considerations

There are four important considerations of eager fetching that you should consider:

- When you are using `parallel` eager fetch mode and you have large result sets enabled (see [Section 4.10, “Large Result Sets” \[180\]](#)) or you place a range on a query, Kodo performs the needed parallel selects on one page of results at a time. For example, suppose your `FetchBatchSize` is set to 20, and you perform a large result set query on a class that has collection fields in the configured fetch groups. Kodo will immediately cache the first 20 results of the query using `join` mode eager fetching only. Then, it will issue the extra selects needed to eager fetch your collection fields according to `parallel` mode. Each select will use a SQL `IN` clause (or multiple `OR` clauses if your class has a compound primary key) to limit the selected collection elements to those owned by the 20 cached results.

Once you iterate past the first 20 results, Kodo will cache the next 20 and again issue any needed extra selects for collection fields, and so on. This pattern ensures that you get the benefits of eager fetching without bringing more data into memory than anticipated.

- Eager fetching can sometimes be *less* efficient than standard fetching when circular relations are included in the configured fetch groups.
- Once Kodo eager-joins into a class, it cannot issue any further eager to-many joins or parallel selects from that class in the same query. To-one joins, however, can recurse to any level.
- Using a to-many join makes it impossible to determine the number of instances the result set contains without traversing the entire set. This is because each result object might be represented by multiple rows. Thus, queries with a range specification or queries configured for lazy result set traversal automatically turn off eager to-many joining.

## 14.3. Datastore Cache

### 14.3.1. Overview of Kodo JDO Datastore Caching

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

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 five ways to access data via the JDO APIs: standard relation traversal, large result set 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 large result set relations or extent iterators. If you find yourself in need of higher-performance extent iteration, see [Example 14.16, “Query Replaces Extent” \[335\]](#)

*Table 14.1. Data access methods*

Access method	Uses cache
Standard relation traversal	Yes
Large result set relation traversal	No
JDOQL query	Yes
<code>PersistenceManager.getObjectById</code>	Yes
Iteration over an extent	No

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 the use of the distributed event notification framework, described in [Section 14.4, “Remote Event Notification Framework” \[335\]](#)

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 `kodo.event.RemoteCommitProvider`, commit information is communicated to other JVMs via JMS or TCP, and remote caches are invalidated based on this information.

See the descriptions of the different remote commit providers in [Section 14.4.1, “Remote Commit Provider Configuration” \[?\]](#) for details on multi-JVM cache synchronization options.

When using a Tangosol Coherence cache plug-in, all remote updating of cache information is delegated to the Coherence cache.

### 14.3.2. Kodo JDO Cache Usage

To enable the basic single-`PersistenceManagerFactory` cache, set the `kodo.DataCache` property to `true`, and set the `kodo.RemoteCommitProvider` property to `sjvm`:

```
kodo.DataCache: true
kodo.RemoteCommitProvider: sjvm
```

To configure the `PersistenceManagerFactory` cache to remain up-to-date in a distributed environment, set the `kodo.RemoteCommitProvider` property appropriately. This process is described in greater depth in [Section 14.4, “Remote Event Notification Framework” \[335\]](#)

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 the `CacheSize` property in your plugin string -- see below for an example. Objects that are pinned into the cache are not counted when determining if the cache size exceeds the maximum.

Expired objects are moved to a soft reference map, so they may stick around for a little while longer. You can control the number of soft references Kodo keeps with the `SoftReferenceSize` property. Soft references are unlimited by default. Set to 0 to disable soft references completely.

```
kodo.DataCache: true(CacheSize=5000, SoftReferenceSize=0)
```

A cache timeout value can be specified for a class by setting the `data-cache-timeout` metadata extension to a positive number representing the amount of time in milliseconds for which a class's data is valid. Use a value of -1 for no expiration. This is the default value.

#### *Example 14.4. Specifying a DataCache Timeout*

```
<class name="Employee">
  <!-- time out employee objects after 10 seconds -->
  <extension vendor-name="kodo" key="data-cache-timeout" value="10000"/>
</class>
```

A specific cache can specify that it should be cleared at specific times instead of invalidating values after a period of time. The default cache implementations can take cron formatted eviction schedule specified as the `EvictionSchedule` property. The format of this property is a white space separated list of five tokens, where the \* symbol (asterisk), indicates match all. The tokens are in order:

- Minute
- Hour of Day
- Day of Month
- Month
- Day of Week

For example, this would schedule the default cache to evict values from the cache at 15 and 45 past 3 PM on Sunday.

```
kodo.DataCache: true(EvictionSchedule="15,45 15 * * 1")
```

It is also possible for different persistence-capable classes to use different caches. This is achieved by specifying a cache name in

the **data-cache** metadata extension.

### ***Example 14.5. Specifying a Non-Default DataCache***

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

This will cause instances of the `Employee` class to be stored in a cache named `small-cache`. This `small-cache` cache can be explicitly configured in the `kodo.DataCache` plugin string, or can be implicitly defined, in which case it will take on the same default configuration properties as the default cache identified in the `kodo.DataCache` property.

### ***Example 14.6. Configuring and Acquiring a Named DataCache***

```
kodo.DataCache: true, true(Name=small-cache, CacheSize=100)
```

```
import kodo.datacache.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
DataCache smallCache = kpm.getConfiguration().getDataCacheManager().
    getDataCache ("small-cache");
```

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

### ***Example 14.7. Pinning an Object into the DataCache***

```
import kodo.datacache.*;
import kodo.runtime.*;

...

DataCache cache = KodoHelper.getDataCache (o);
cache.pin (JDOHelper.getObjectId (o));
```

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

### ***Example 14.8. Unpinning an Object from the DataCache***

```
import kodo.datacache.*;
import kodo.runtime.*;

...

DataCache cache = KodoHelper.getDataCache (o);
cache.unpin (JDOHelper.getObjectId (o));
```

It is also possible to explicitly evict data from the cache.

### ***Example 14.9. Evicting an Object from the DataCache***

```
import kodo.datacache.*;
import kodo.runtime.*;

...

DataCache cache = KodoHelper.getDataCache (o);
cache.remove (JDOHelper.getObjectId (o));
```

Rather than evicting objects from the data cache directly, you can also configure Kodo to automatically evict objects from the data cache when you use the persistence manager's eviction APIs.

### ***Example 14.10. Data Cache Eviction Through the Persistence Manager***

```
kodo.PersistenceManagerImpl: EvictFromDataCache=true
```

---

## **14.3.3. 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 `CacheSize` plugin property. Like the data cache, the query cache also has a backing soft reference map. The `SoftReferenceSize` property controls the size of this map. It defaults to no limit.

### ***Example 14.11. Setting the Size of the Query Cache***

```
kodo.QueryCache: CacheSize=1000, SoftReferenceSize=0
```

To disable the query cache completely, set the `kodo.QueryCache` property to `false`:

### ***Example 14.12. Disabling the Query Cache***

```
kodo.QueryCache: false
```

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.
- Caching is not used when the the data cache does not have any cached data for an ID in a query result.
- Queries that use custom result classes, groupings, aggregates, or projections are not cached.

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.

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.

### ***Example 14.13. Notifying the Query Cache of Altered Classes***

```
import kodo.datacache.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
QueryCache cache = kpm.getConfiguration().getDataCacheManager().
    getQueryCache();
Class[] changed = new Class[] { A.class, B.class };
cache.classesChanged (Arrays.asList (changed));
```

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 14.14. Dropping or Pinning Query Results***

```
import kodo.datacache.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
QueryCache cache = kpm.getConfiguration ().getDataCacheManager ().
    getQueryCache ();

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

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

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-persistence manager or per-query basis:

### *Example 14.15. Disabling and Enabling Query Caching*

```
import kodo.query.*;
import kodo.runtime.*;

...

// temporarily disable query caching for all queries created from pm
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getFetchConfiguration ().setQueryCacheEnabled (false);

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

---

## 14.3.4. DataCache Integrations

Several integrations to third party cache solutions for Kodo's JDO data cache exist. For details, see [Appendix F, DataCache Integrations \[428\]](#)

---

## 14.3.5. Cache Extension

The provided data cache classes can be easily extended to add additional functionality. If you are adding new behavior, you should extend `kodo.datacache.DataCacheImpl`. To use your own storage mechanism, extend `kodo.datacache.AbstractDataCache`, or implement `kodo.datacache.DataCache` directly. If you want to implement a distributed cache that uses an unsupported method for communications, create an implementation of `kodo.event.RemoteCommitProvider`. This process is described in greater detail in [Section 14.4.2, “Customization” \[?\]](#).

The query cache is just as easy to extend. Add functionality by extending the default `kodo.datacache.QueryCacheImpl`. Implement your own storage mechanism for query results by extending `kodo.datacache.AbstractQueryCache` or implementing the `kodo.datacache.QueryCache` interface directly.

---

## 14.3.6. Important Notes

- The default cache implementations *do not* automatically refresh objects in other persistence managers when the cache is updated or invalidated. This behavior would not be compliant with the JDO specification.
- Invoking `PersistenceManager.evict` *does not* result in the corresponding data being dropped from the data cache, unless you have set the proper configuration options as explained above (see [Example 14.10, “Data Cache Eviction Through the Persistence Manager”](#) [332]). Other methods related to the persistence manager cache also do not effect the datastore cache. The datastore cache 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. To manipulate the datastore cache, you should generally use its APIs directly.
- A `kodo.event.RemoteCommitProvider` must be specified (via the `kodo.RemoteCommitProvider` property) in order to use the data cache, even when using the cache in a single-JVM mode. When using it in a single-JVM context, the property can be set to `sjvm`.

### 14.3.7. 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 14.16. Query Replaces Extent*

```
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 ());
```

- Queries that use parameters that are FCOs are not cached.

## 14.4. Remote Event Notification Framework

---

The remote event notification framework allows a subset of the information available through Kodo JDO's **transaction event**

**system** to be broadcast to remote listeners. The **L2 data cache**, for example, uses remote events to remain synchronized when deployed in multiple JVMs.

To enable remote events, you must configure the `PersistenceManagerFactory` to use a `RemoteCommitProvider` (see below). Given that a `RemoteCommitProvider` is properly configured, it is possible to register **RemoteCommitListeners** that will be alerted with a list of committed object ids whenever a transaction on a remote machine successfully commits.

## 14.4.1. Remote Commit Provider Configuration

Kodo JDO includes built in remote commit providers for JMS and TCP communication. The JMS remote commit provider can be configured by setting the `kodo.RemoteCommitProvider` to contain the appropriate configuration properties. The JMS provider understands the following properties:

- **Topic:** The topic that the remote commit provider should publish notifications to and subscribe to for notifications sent from other JVMs. Defaults to `topic/KodoCommitProviderTopic`
- **TopicConnectionFactory:** The JNDI name of a `javax.jms.TopicConnectionFactory` factory to use for finding topics. Defaults to `java:/ConnectionFactory`. This setting may vary depending on the application server in use; consult the application server's documentation for details of the default JNDI name for the `javax.jms.TopicConnectionFactory` instance. For example, under Weblogic, the JNDI name for the `TopicConnectionFactory` is `"javax.jms.TopicConnectionFactory"`.
- **ExceptionReconnectAttempts:** The number of times to attempt to reconnect if the JMS system notifies Kodo of a serious connection error. Defaults to 0, meaning Kodo will log the error but otherwise ignore it, hoping the connection is still valid.
- **\***: All other configuration properties will be interpreted as settings to pass to the JNDI `InitialContext` on construction. For example, you might set the `java.naming.provider.url` property to the URL of the context provider.

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

### *Example 14.17. JMS Remote Commit Provider Configuration*

```
kodo.RemoteCommitProvider: jms(Topic=topic/KodoCommitProviderTopic)
```

### Note

Because of the nature of JMS, it is important that you invoke `PersistenceManagerFactory.close` when finished with a persistence manager factory. If you do not do so, a daemon thread will stay up in the JVM, preventing the JVM from exiting.

The TCP remote commit provider 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 `kodo.RemoteCommitProvider` plugin property to contain the appropriate configuration settings. The TCP provider understands the following properties:

- **Port:** The TCP port that the provider should listen on for commit notifications. Defaults to 5636.
- **Addresses:** A semicolon-separated list of IP addresses to which notifications should be sent. No default value.

To configure a persistence manager factory to use the TCP provider, your properties filename might look like the following:

### *Example 14.18. TCP Remote Commit Provider Configuration*

```
kodo.RemoteCommitProvider: tcp(Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13)
```

## 14.4.2. Customization

Additional mechanisms for remote event notification can be easily developed by creating an implementation of the **RemoteCommitProvider** interface, possibly by extending the **AbstractRemoteCommitProvider** abstract class. For details on particular customization needs, contact SolarMetric at [jdossupport@solarmetric.com](mailto:jdossupport@solarmetric.com).

## 14.5. Fetch Groups

The JDO specification defines a concept of a default fetch group, but it does not touch upon additional, non-default fetch groups. 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. Specifying fetch groups allows for tuning of lazy loading and eager fetching behavior.

### 14.5.1. Normal Default Fetch Group Behavior

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

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

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

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.

## 14.5.2. 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 (unless configured otherwise; see the next paragraph), 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 a Kodo persistence manager, query, or extent to use a particular fetch group or set of fetch groups when loading new objects, as described later in this chapter. When this is the case, 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:

### *Example 14.19. Custom Fetch Group Meta-Data*

```
<?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>

```

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 persistence manager 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.

### 14.5.3. Custom Fetch Group Configuration

You can control the default set of fetch groups Kodo JDO will load when initializing a persistent object through the `kodo.FetchGroups` configuration property. Set this property to a comma-separated list of fetch group names.

As mentioned above, it is also possible to override the global `FetchGroups` property on individual Kodo persistence managers, queries, and extents. All of these components give you access to their internal `FetchConfiguration` object for controlling object loading behavior. [Chapter 10, JDO Runtime Extensions \[288\]](#) details these interfaces, including the `FetchConfiguration` in [Section 10.5, “Fetch Configuration” \[290\]](#).

#### *Example 14.20. Adding a Fetch Group to a Query*

```

import kodo.query.*;

...

KodoQuery kq = (KodoQuery) pm.newQuery (FetchGroupExample.class, "a > 5");
kq.getFetchConfiguration ().addFetchGroup ("g1");
Collection results = (Collection) kq.execute ();

```

### 14.5.4. Per-field Fetch Configuration

In addition to controlling fetch configuration on a per-fetch-group basis, you can configure Kodo to include certain particular fields in the current fetch configuration. This allows you to add individual fields that are not in the default fetch group or in any other currently-active fetch groups to the set of fields that will be eagerly loaded from the database. Per-field control of the fetch configuration happens through the `FetchConfiguration` interface.

#### *Example 14.21. Adding a Single Field to a PersistenceManager*

```

import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getFetchConfiguration ().addField (Person.class, "firstName");

```

The field named in the method invocation must be defined in the class specified in the invocation. This applies for inheritance hierarchies. So, if the field `firstName` is defined in class `Person`, you must invoke `addField (Person.class, "firstName")` and not `addField (Employee.class, "firstName")`. This is stricter than Java's default field-masking algorithms, which would allow the latter method behavior if `Employee` did not also define a field called `firstName`.

In order to avoid the cost of reflection, Kodo does not perform any validation of the field name / class name pairs that you put into the fetch configuration. If you specify nonexistent class / field pairs, nothing adverse will happen, but you will receive no notification of the fact that the specified configuration is not being used.

The `FetchConfiguration` interface provides convenience methods to add more than one field at a time and to use a single fully-qualified field name rather than a class and a field name.

## 14.6. Lock Groups

The JDO specification provides support for both optimistic and pessimistic transaction types. Typically, optimistic locking is performed at the object level of granularity. That is, concurrent changes in different transactions to any part of a common object will result in an optimistic locking exception being thrown in the transaction that commits last. In many applications, this is acceptable. However, if your application has a high likelihood of concurrent writes to different parts of the same object, then it may be advantageous to use a finer-grained optimistic lock. Additionally, certain parts of an object model may be best modelled without any locking at all, or with a last-commit-wins strategy. It is for these types of situations that Kodo offers customizable optimistic lock groups, which allow you to achieve sub-object-level locking granularity.

For example, an `Employee` class may have some fields configurable by the employee that the object represents (`firstName`, `lastName`, `phoneNumber`), some that are only modifiable by that employee's manager (`salary`, `title`), and some in which concurrent updates are acceptable (a list of projects). In such a model, you can greatly improve the success of concurrent updates in optimistic transactions by putting `firstName`, `lastName`, and `phoneNumber` into one lock group, `salary` and `title` into another, and excluding the `projects` field from optimistic lock checks altogether:

### *Example 14.22. Lock Group Meta-Data*

```
public class Employee
{
    // these fields are configurable by this employee
    private String      firstName;
    private String      lastName;
    private String      phoneNumber;

    // these fields can only be set by the employee's manager
    private float       salary;
    private String      title;

    // this field might be updated concurrently by the employee,
    // other team members, or the employee's manager
    private Set         projects;
}

<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="Employee">
      <!-- these fields are implicitly left in the default lock group -->
      <field name="firstName"/>
      <field name="lastName"/>

      <!-- this field is explicitly put into the default lock group -->
      <field name="phoneNumber">
        <extension vendor-name="kodo" key="lock-group" value="default"/>
      </field>

      <!-- these are explicitly put into a non-default lock group -->
      <field name="salary">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <field name="title">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
    </class>
  </package>
</jdo>
```

```

    <!-- concurrent updates to this field are ok, so it's
    excluded from locking -->
    <field name="projects">
      <collection element-type="Project"/>
      <extension vendor-name="kodo" key="lock-group" value="none"/>
    </field>
  </class>
</package>
</jdo>

```

Currently, lock groups are only supported when using a relational database, either through a direct connection to the database or through a remote persistence manager. Additionally, lock groups are only currently available when using the `version-number` or `version-date` version indicators (that is, lock groups are not supported by the `state-image` version indicator). Additionally, lock groups should transparently work with any custom extension of `kodo.jdbc.meta.ColumnVersionIndicator`.

## 14.6.1. Lock Groups and Subclasses

Custom lock groups are a bit limited when it comes to subclass fields. Because of the way that lock groups are parsed, subclasses cannot simply declare additional lock groups implicitly, as is done in the example shown above. Instead, the least-derived type that the subclass inherits from must list all lock groups that its children can use via the class-level `lock-groups` metadata extension. For example, if the `Employee` class in the last example extended `Person`, the metadata would have looked like so:

### *Example 14.23. Lock Group Meta-Data*

```

public class Person
{
    // these fields are configurable by this person
    private String      firstName;
    private String      lastName;
    private String      phoneNumber;
}

public class Employee
    extends Person
{
    // these fields can only be set by the employee's manager
    private float       salary;
    private String      title;

    // this field might be updated concurrently by the employee,
    // other team members, or the employee's manager
    private Set         projects;
}

<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="Person">
      <!-- here we list the lock groups that will be used by Employee. -->
      <extension vendor-name="kodo" key="lock-groups" value="corporate"/>

      <!-- these fields are implicitly left in the default lock group -->
      <field name="firstName"/>
      <field name="lastName"/>

      <!-- this field is explicitly put into the default lock group -->
      <field name="phoneNumber">
        <extension vendor-name="kodo" key="lock-group" value="default"/>
      </field>
    </class>

    <class name="Employee">
      <!-- these are explicitly put into a non-default lock group -->
      <field name="salary">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <field name="title">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
    </class>
  </package>
</jdo>

```

```

    <!-- concurrent updates to this field are ok, so it's
         excluded from locking -->
    <field name="projects">
      <collection element-type="Project"/>
      <extension vendor-name="kodo" key="lock-group" value="none"/>
    </field>
  </class>
</package>
</jdo>

```

The exceptions to this rule are the `none` and `default` built-in lock groups. They can be used at any point in the inheritance hierarchy without pre-declaration. Additionally, the `lock-groups` listing can contain lock groups that would otherwise be implicitly defined in the least-derived type metadata.

## 14.6.2. Lock Group Mapping

When using the custom lock groups with a relational database, Kodo will need a lock column for each of the groups, instead of just one lock column. (Of course, no lock column will be needed for the special `none` group.) Kodo currently requires that all the lock columns for a given object be in the same table. Additionally, it is only possible to use a single version indicator strategy for a given object. That is, you cannot have one numeric lock column and another date lock column.

### *Example 14.24. Using Lock Groups with version-number Indicator*

For the example presented above, corresponding mapping information might look like so. Note that the lock group name is prepended to the attribute name / extension value when specifying the column name for use by non-default lock groups.

```

Schema:

<table name="EMPLOYEE">
  ... primary key columns
  <column name="VERSION1" type="bigint"/>
  <column name="VERSION2" type="bigint"/>
  ... columns for magazine fields ...
</table>

Mapping information using the mapping XML format:

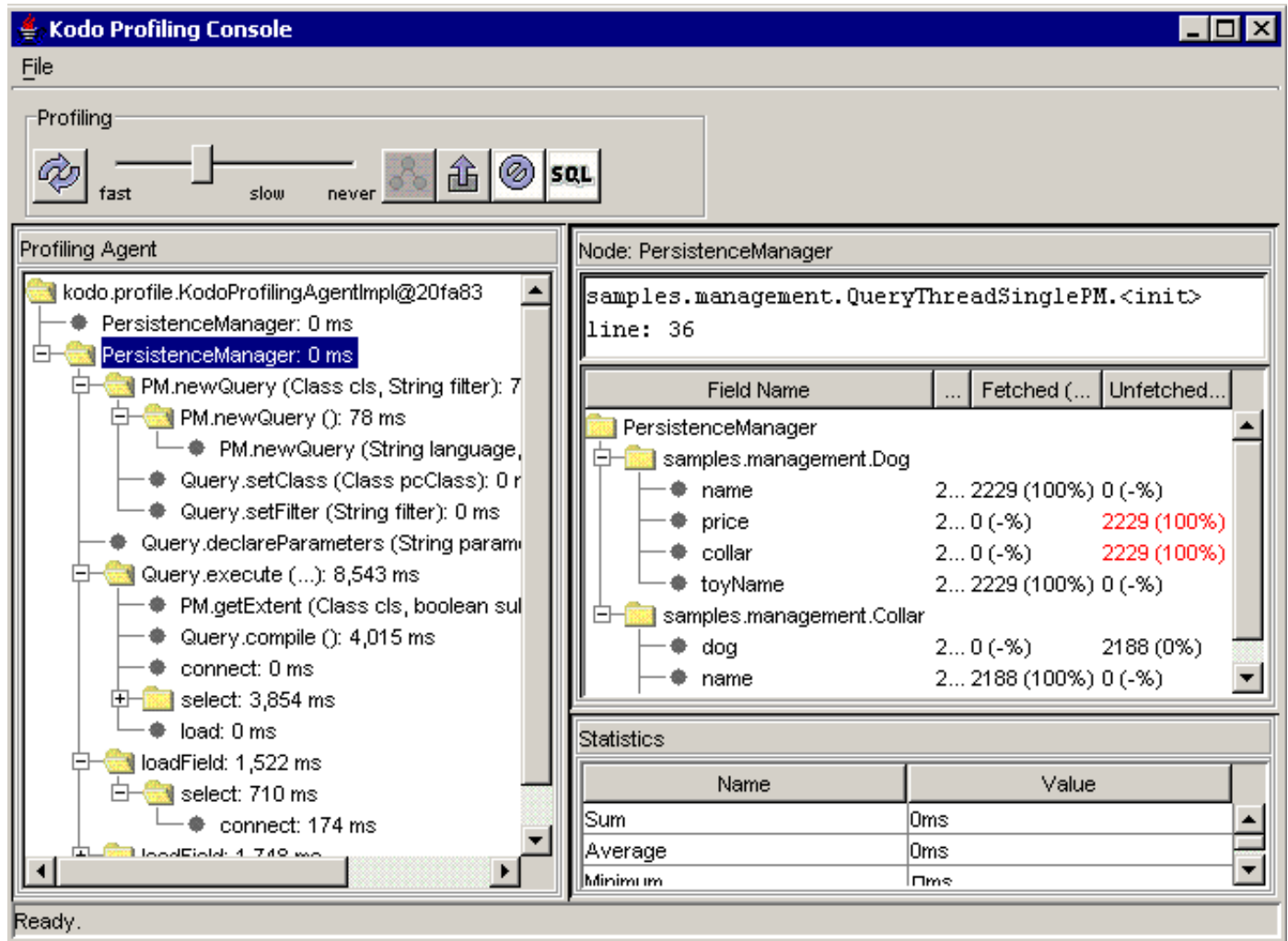
<class name="Employee">
  ... class mapping ...
  <jdbc-version-ind type="version-number" column="VERSION1"
    corporate-column="VERSION2"/>
  ... field mappings ...
</class>

Mapping information using JDO metadata extensions:

<class name="Employee">
  ... class extensions ...
  <extension vendor-name="kodo" key="jdbc-version-ind" value="version-number">
    <extension vendor-name="kodo" key="column" value="VERSION1"/>
    <extension vendor-name="kodo" key="corporate-column" value="VERSION2"/>
  </extension>
  ... field metadata ...
</class>

```

## 14.7. Profiling



The Kodo profiling capability allows for profiling of application code. It is designed to help optimize the use of Kodo, and is not intended to be a generic profiling tool. Only Kodo specific APIs are instrumented.

The profiling capability can either be used standalone using the Kodo Profiling Console, or inside the Kodo Management Console using the Profiling MBean. To use the profiling capability within the Kodo Management Console see [Section 12.4.8, “Profiling MBean” \[319\]](#).

### 14.7.1. Profiling in an embedded GUI

The profiling capability can be used standalone locally. To bring up the Kodo Profiling Console, set the following property (see [Section 2.6.27, “kodo.ManagementConfiguration” \[151\]](#)):

- `kodo.ManagementConfiguration: profiling-gui`

The left pane of the Profiling Console contains a tree. Each node in the tree represents a call in a call stack. The root of each call stack is a `PersistenceManager`. Note that you must use the default `PersistenceManager` (or an extension of it) in order to see profiling information. Each node may be composed of three items - the node name, the amount of time spent in each node, and the percentage of the total time spent in the parent node that this child node contributes. Details about a node can be seen in the right panes by selecting a node. The top right pane(s) yield detailed information about the node (e.g. the location in code where a transaction was started, or the SQL generated for a query), and the bottom right pane contains statistics about the node.

The Profiling Console has a Profiling Toolbar. It has the following controls:

- `Refresh`: Refreshes the statistics shown in the tree.
- `Refresh slider`: Set the interval at which the tree will be refreshed.
- `Show descendents`: Show the tree with `PersistenceManagers` as roots. This is the normal view. This option is available only when the `Show ancestors` view is in use. See the `Show ancestors` feature below.
- `Export . . .`: Export the current profiling data to a file for later viewing. Exported profiling data is stored in files ending in `.prx`. The export can be viewed using the `profilingviewer` application. To do this run the `profilingviewer` application and pass in the name of the exported file.
- `Reset`: Resets the statistics shown in the tree.
- `Show SQL`: Show an inverted tree where the root nodes are SQL statements, and their ancestors are shown as children in the tree.

The profiling call tree has a context menu that is brought up in an operating system dependent manner (e.g. right click on Windows). The following options are available:

- `Refresh`: Refreshes the statistics shown in the tree.
- `Show ancestors`: Show an inverted tree where the nodes of the same name and description as the selected node are used as the roots, and their ancestors are shown as children in the tree.
- `Show descendents`: Show the tree with `PersistenceManagers` as roots. This is the normal view. This option is available only when the `Show ancestors` view is in use.
- `Export . . .`: Export the current profiling data to a file for later viewing. Exported profiling data is stored in files ending in `.prx`. The export can be viewed using the `profilingviewer` application. To do this run the `profilingviewer` application and pass in the name of the exported file.
- `Reset`: Resets the statistics shown in the tree.

The detail information for a `PersistenceManager` contains information about objects that were fetched in the context of that `PersistenceManager`. The detail area consists of information about where in code the `PersistenceManager` was created, and a `TreeTable` with information on each field in each `Persistence Capable` class. The `TreeTable` has the following columns:

- `Field Name`: The name of the persistent field.
- `Total`: The total number of times the object containing that field was loaded in the context of the containing `PersistenceManager`.
- `Fetched / % Used`: The first number represents the number of times the field was fetched during the initial load. The second number represents the percentage of initially fetched fields that are actually accessed. A low percentage indicates that perhaps that field should not be in the default fetch group or in a fetch group configured for initial load.
- `Unfetched / % Used`: The first number represents the number of times the field was NOT fetched during the initial load. The second number represents the percentage of the time the initially unfetched field is actually accessed. A high percentage indicates that perhaps that field should be in the default fetch group or in a fetch group configured for initial load.

## 14.7.2. Dumping profiling data to disk from a batch process

---

The profiling capability can be used to create profiling exports. Exported profiling data is stored in files ending with the `.prx` suffix. To view exported profiling data, use the `profilingviewer` command, e.g. `profilingviewer myexport.prx`. To enable automatic export of profiling data, set the following properties (see [Section 2.6.27](#), “**kodo.ManagementConfiguration**” [151]):

- `kodo.ManagementConfiguration: profiling-export`

When exporting, the `ManagementConfiguration` value takes the following optional parameters:

- `intervalMillis`: The number of milliseconds between exports (defaults to -1, indicating that there will be a single export upon exit).
- `basename`: The basename of the exported data file to create.
- `uniqueNames`: A boolean that indicates whether or not the exported data file name should have the systems current time in milliseconds included as part of the name in order to make it unique.

For example, in order to export data every five minutes with a basename of `MyExport` include in your properties the following line:

- `kodo.ManagementConfiguration: profiling-export(intervalMillis=300000,basename="MyExport")`

## 14.7.3. Controlling how the profiler obtains context information

---

The description of `PersistenceManager` and transaction nodes are dependent on the call stack when the `PersistenceManager` is created, or the transaction is started, and can be controlled by an optional parameter to the `kodo.ManagementConfiguration` property (see [Section 2.6.27](#), “**kodo.ManagementConfiguration**” [151]):

- `StackStyle`: Indicates how much of the call stack where the `PersistenceManager` or transaction was created/started to include. "line" indicates a single line, "partial" indicates a partial stack starting at the user code, and "full" indicates a full call stack, including Kodo code. Defaults to "line". The full call stack is only interesting from a debugging standpoint.

---

# Chapter 15. Third Party Integration

## 15.1. 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.

## 15.2. 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 pre-built Ant task definitions for all bundled tools:

- **JDO Enhancer Task**
- **Application Identity Tool Task**
- **JDO Metadata Tool Task**
- **Mapping Tool Task**
- **Reverse Mapping Tool Task**
- **Schema Tool Task**
- **Schema Generator Task**

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

### 15.2.1. Common Ant Configuration Options

---

All Kodo tasks accept 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 15.1. Using the `<config>` Ant Tag*

```
<mappingtool action="refresh">
  <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" />
</mappingtool>
```

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

### ***Example 15.2. Using the Properties Attribute of the <config> Tag***

```
<mappingtool action="refresh">
  <fileset dir="${basedir}">
    <include name="**/*.jdo"/>
  </fileset>
  <config properties="kodo-dev.properties"/>
</schematool>
```

### ***Example 15.3. Using the PropertiesFile Attribute of the <config> Tag***

```
<mappingtool action="refresh">
  <fileset dir="${basedir}">
    <include name="**/*.jdo"/>
  </fileset>
  <config propertiesFile="../conf/kodo-dev.properties"/>
</schematool>
```

Tasks 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 so the enhancer and mapping tool can locate the relevant classes.

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

### ***Example 15.4. Using the <classpath> Ant Tag***

```
<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>
```

Finally, tasks that invoke code-generation tools like the application identity tool and reverse mapping tool accept a nested `<codeformat>` element. See the [code formatting documentation](#) for a list of code formatting attributes.

### ***Example 15.5. Using the <codeformat> Ant Tag***

```
<reversemappingtool package="com.xyz.jdo" directory="${basedir}/src">
  <codeformat tabSpaces="4" spaceBeforeParen="true" braceOnSameLine="false"/>
</reversemappingtool>
```

## 15.2.2. JDO Enhancer Ant Task

---

The JDO enhancer task allows you to invoke the JDO enhancer directly from within the Ant build process. It takes a nested `<fileset>` tag to specify the files that should be processed. You can specify `.java`, `.jdo`, or `.class` files. The task's parameters correspond exactly to the long versions of the command-line arguments to `jdoc`.

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

### *Example 15.6. Invoking the JDO Enhancer from Ant*

```
<target name="enhance">
  <!-- define the jdoc task; this can be done at the top of the      -->
  <!-- build.xml file, so it will be available for all targets      -->
  <taskdef name="jdoc" classname="kodo.ant.JDOEnhancerTask"/>

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

## 15.2.3. Application Identity Tool Ant Task

---

The application identity tool task allows you to invoke the application identity tool directly from within the Ant build process. It takes a nested `<fileset>` tag to specify the files that should be processed. You can specify `.java`, `.jdo`, or `.class` files. The task's parameters correspond exactly to the long versions of the command-line arguments to the `appidtool`.

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

### *Example 15.7. Invoking the Application Identity Tool from Ant*

```
<target name="appids">
  <!-- define the appidtool task; this can be done at the top of    -->
  <!-- the build.xml file, so it will be available for all targets  -->
  <taskdef name="appidtool" classname="kodo.ant.ApplicationIdToolTask"/>

  <!-- invoke tool on all .jdo files below the current directory    -->
  <appidtool>
    <fileset dir=".">
      <include name="**/*.jdo" />
    </fileset>
    <codeformat spaceBeforeParen="true" braceOnSameLine="false"/>
  </appidtool>
</target>
```

## 15.2.4. JDO Metadata Tool Ant Task

---

The JDO metadata tool task allows you to invoke the JDO metadata tool directly from within the Ant build process. We do not recommend that you regenerate your JDO metadata often, but the task is available nonetheless. It takes a nested `<fileset>` tag to specify the files that should be processed. You can specify `.java` or `.class` files. The task's parameters correspond exactly to the long versions of the command-line arguments to the `metadatatool`.

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

### *Example 15.8. Invoking the JDO Metadata Tool from Ant*

```
<target name="genmetadata">
  <!-- define the metadatatool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="metadatatool" classname="kodo.ant.JDOMetaDataToolTask"/>

  <!-- invoke tool on all *PC.java files in the current directory -->
  <metadatatool file="package.jdo">
    <fileset dir=".">
      <include name="*PC.java"/>
    </fileset>
  </metadatatool>
</target>
```

## 15.2.5. Mapping Tool Ant Task

---

The mapping tool task allows you to directly invoke the mapping tool from within the Ant build process. It is useful for making sure that the database schema and object-relational mapping data is always synchronized with your persistent class definitions, without needing to remember to invoke the mapping tool manually. The task's parameters correspond exactly to the long versions of the command-line arguments to the `mappingtool`.

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

### *Example 15.9. Invoking the Mapping Tool from Ant*

```
<target name="refresh">
  <!-- define the mappingtool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="mappingtool" classname="kodo.jdbc.ant.MappingToolTask"/>

  <!-- add the schema components for all .jdo files below the -->
  <!-- current directory -->
  <mappingtool action="refresh">
    <fileset dir=".">
      <include name="**/*.jdo" />
    </fileset>
  </mappingtool>
</target>
```

## 15.2.6. Reverse Mapping Tool Ant Task

---

The reverse mapping tool task allows you to directly invoke the reverse mapping tool from within Ant. While many users will only run the reverse mapping process once, others will make it part of their build process. The task's parameters correspond exactly to the long versions of the command-line arguments to the `reversemappingtool`.

Following is an example of a `build.xml` target that invokes the reverse mapping tool:

***Example 15.10. Invoking the Reverse Mapping Tool from Ant***

```
<target name="reversemap">
  <!-- define the reversemappingtool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="reversemappingtool"
    classname="kodo.jdbc.ant.ReverseMappingToolTask"/>

  <!-- reverse map the entire database -->
  <reversemappingtool package="com.xyz.jdo" directory="${basedir}/src"
    customizerProperties="${basedir}/conf/reverse.properties">
    <codeformat tabSpaces="4" spaceBeforeParen="true" braceOnSameLine="false"/>
  </reversemappingtool>
</target>
```

---

## 15.2.7. Schema Tool Ant Task

The schema tool task allows you to directly invoke the schema tool from within the Ant build process. The task's parameters correspond exactly to the long versions of the command-line arguments to the `schematool` .

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

***Example 15.11. Invoking the Schema Tool from Ant***

```
<target name="schema">
  <!-- define the schematool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="schematool" classname="kodo.jdbc.ant.SchemaToolTask"/>

  <!-- add the schema components for all .schema files below the -->
  <!-- current directory -->
  <schematool action="add">
    <fileset dir=".">
      <include name="**/*.schema" />
    </fileset>
  </schematool>
</target>
```

---

## 15.2.8. Schema Generator Ant Task

The schema generator task allows you to directly invoke the schema tool from within the Ant build process. The task's parameters correspond exactly to the long versions of the command-line arguments to `schemagen` .

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

***Example 15.12. Invoking the Schema Generator from Ant***

```
<target name="genschema">
  <!-- define the schemagen task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="schemagen" classname="kodo.jdbc.ant.SchemaGeneratorTask"/>
```

```
<!-- generate the database schema to the schema.xml file -->
<schemagen file="${basedir}/schema.xml"/>
</target>
```

## 15.3. 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. The latest versions of XDoclet include JDO doclet that can create JDO metadata files.

In order to use XDoclet, you will need to download the XDoclet libraries separately from <http://xdoclet.sourceforge.net> and add the `xdoclet-<version>.jar`, `xdoclet-jdo-module-<version>.jar`, and `xjavadoc-<version>.jar` libraries to your CLASSPATH. Kodo has been tested with XDoclet version 1.2b3 and XJavadoc version 1.0. Any higher version should work as well.

The example below shows how to comment your code with XDoclet JDO tags. The source code should be self-explanatory.

### *Example 15.13. Commenting for XDoclet*

```
package samples.xdoclet;

import java.util.*;

/**
 * <p>This is a simple example class using XDoclet. It is used to demonstrate
 * automatic generation of the JDO Metadata file based on the
 * <code>jdo.*</code> XDoclet tags in the source code.</p>
 *
 * <p>The JDO tags above the class element apply to class-level metadata. The
 * <code>jdo.persistence-capable</code> tag is required to denote a persistent
 * class. The tag takes the same attributes as the <code>class</code> element
 * in standard JDO metadata. You can include class-level vendor extensions
 * with the <code>jdo.class-vendor-extension</code> tag.</p>
 *
 * @jdo.persistence-capable
 *     identity-type="application"
 *     objectid-class="Main$Id"
 * @jdo.class-vendor-extension
 *     vendor-name="kodo"
 *     key="data-cache-timeout"
 *     value="10"
 */
public class Main
{
    /**
     * Field-level metadata is declared with the <code>jdo.field</code> tag.
     * It takes the same attributes as the <code>field</code> element in
     * standard JDO metadata.
     *
     * @jdo.field
     *     primary-key="true"
     * @jdo.field-vendor-extension
     *     vendor-name="kodo"
     *     key="jdbc-auto-increment"
     *     value="true"
     */
    private String pk1;

    /**
     * You are not required to place all attributes on a separate line as
     * we have been doing above.
     *
     * @jdo.field primary-key="true"
     * @jdo.field-vendor-extension vendor-name="kodo"
     *     key="jdbc-size" value="20"
     */
    private String pk2;

    /**
```

```
* @jdo.field
*   null-value="exception"
*   default-fetch-group="false"
*/
private String name;

private Main main;

/**
 * @jdo.field
 *   default-fetch-group="true"
 *   collection-type="collection"
 *   element-type="Main"
 *   embedded-element="false"
 * @jdo.field-vendor-extension
 *   vendor-name="kodo"
 *   key="inverse-owner"
 *   value="main"
 */
private Collection nodes = new ArrayList ();

/**
 * @jdo.field
 *   collection-type="map"
 *   key-type="String"
 *   value-type="Integer"
 */
private Map cache = new HashMap ();

public String getPk1 ()
{
    return this.pk1;
}

public void setPk1 (String pk1)
{
    this.pk1 = pk1;
}

public String getPk2 ()
{
    return this.pk2;
}

public void setPk2 (String pk2)
{
    this.pk2 = pk2;
}

public String getName ()
{
    return this.name;
}

public void setName (String name)
{
    this.name = name;
}

public Main getMain ()
{
    return this.main;
}

public void setMain (Main main)
{
    this.main = main;
}

public Collection getNodes ()
{
    return this.nodes;
}

public Map getCache ()
{
    return this.cache;
}

/**
 * Application identity class.
 */
```

```

public static class Id
{
    private static final char DELIM = '/';

    public String pk1;
    public String pk2;

    public Id ()
    {
    }

    public Id (String serialized)
    {
        int idx = serialized.indexOf (DELIM);
        pk1 = serialized.substring (0, idx);
        pk2 = serialized.substring (idx + 1);
    }

    public boolean equals (Object other)
    {
        if (other == this)
            return true;
        if (!(other instanceof Id))
            return false;

        Id id = (Id) other;
        return pk1.equals (id.pk1) && pk2.equals (id.pk2);
    }

    public int hashCode ()
    {
        return pk1.hashCode () + pk2.hashCode ();
    }

    public String toString ()
    {
        return pk1 + DELIM + pk2;
    }
}

```

XDoclet does not include direct support for nested vendor extensions, which are required to perform O/R mapping in metadata. Instead, Kodo allows you to simulate nested extensions by specifying key attribute paths separated by the / character. For example, to simulate the following class declaration and extensions:

```

<class name="Magazine">
  <extension vendor-name="kodo" key="jdbc-class-map" value="base">
    <extension vendor-name="kodo" key="table" value="MAG"/>
    <extension vendor-name="kodo" key="pk-column" value="ID"/>
  </extension>
  <extension vendor-name="kodo" key="jdbc-version-ind" value="version-number">
    <extension vendor-name="kodo" key="column" value="JDOVERSION"/>
  </extension>
  ...
</class>

```

You could use XDoclet comments as follows:

```

/**
 * @jdo.persistence-capable
 *
 * @jdo.class-vendor-extension
 *   vendor-name="kodo" key="jdbc-class-map" value="base"
 * @jdo.class-vendor-extension
 *   vendor-name="kodo" key="jdbc-class-map/table" value="MAG"
 * @jdo.class-vendor-extension
 *   vendor-name="kodo" key="jdbc-class-map/pk-column" value="ID"
 *
 * @jdo.class-vendor-extension
 *   vendor-name="kodo" key="jdbc-version-ind" value="version-number"

```

```
* @jdo.class-vendor-extension
*   vendor-name="kodo" key="jdbc-version-ind/column" value="JDOVERSION"
*/
```

XDoclet can only be invoked from Ant 1.5 or higher. You can download Ant at <http://jakarta.apache.org/ant/>. XDoclet also requires Log4J, which ships with Kodo.

The example below shows an ant task you might use to invoke XDoclet. Pay close attention to the comments in the source, as XDoclet is picky about how you invoke it.

### Example 15.14. Invoking XDoclet with Ant

```
<taskdef name="jdodoclet" classname="xdoclet.modules.jdo.JdoDocletTask"/>
<target name="xdoclet">
  <echo>
=====
Generating .jdo files from all .java files
=====
  </echo>

  <!-- jdoclet seems to require that the embedded fileset's -->
  <!-- dir attribute is set to the root of your classpath, -->
  <!-- which in this example we're assuming is ${basedir} -->
  <jdodoclet destdir="${basedir}">
    <fileset dir="${basedir}">
      <include name="samples/xdoclet/*.java"/>
    </fileset>

    <!-- this inner task is required to generate metadata; -->
    <!-- the project attribute specifies the name of the -->
    <!-- generated .jdo file; the "generation" attribute -->
    <!-- should be set to "project" to generate single -->
    <!-- "package.jdo" files per-package, or "class" to -->
    <!-- generate metadata on a per-class basis, such as -->
    <!-- "MyClass.jdo" -->
    <jdometadata project="package" generation="project"/>
  </jdodoclet>
</target>
```

Running the above task on the Main shown in our previous example will produce a package.jdo file like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="samples.xdoclet">
    <class name="Main"
      identity-type="application"
      objectid-class="Main$Id"
    > <!-- end class tag -->
    <extension vendor-name="kodo"
      key="data-cache-timeout"
      value="10">
    </extension>
    <field name="pk1"
      primary-key="true"
    > <!-- end field tag -->
    <extension vendor-name="kodo"
      key="jdbc-auto-increment"
      value="true">
    </extension>
    </field>
    <field name="pk2"
      primary-key="true"
    > <!-- end field tag -->
    <extension vendor-name="kodo"
      key="jdbc-size"
      value="20">
    </extension>
  </package>
</jdo>
```

```

    </field>
    <field name="name"
          null-value="exception"
          default-fetch-group="false"
    > <!-- end field tag -->
    </field>
    <field name="nodes"
          default-fetch-group="true"
    > <!-- end field tag -->
        <collection
          element-type="samples.xdoclet.Main"
          embedded-element="false"
        > <!-- end collection tag -->
        </collection>
    <extension vendor-name="kodo"
              key="inverse-owner"
              value="main">
    </extension>
    </field>
    <field name="cache"
    > <!-- end field tag -->
        <map
          key-type="java.lang.String"
          value-type="java.lang.Integer"
        > <!-- end map tag -->
        </map>
    </field>
  </class>
</package>

<!--
To use additional vendor extensions, create a vendor-extensions.xml file that
contains the additional extensions (in extension tags) and place it in your
projects merge dir.
-->

</jdo>

```

A complete XDoclet sample is included in the `samples/xdoclet` directory of your Kodo distribution.

## 15.4. 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), configure mappings, and automatically run the JDO Enhancer.

### 15.4.1. Installing Kodo Into JBuilder

#### Note

We highly recommend fully uninstalling previous versions of the plugin.

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 exten-

sions", then you do not need to perform the manually file copying or any other additional steps.

### Warning

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

## 15.4.2. 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`. The configuration panel provides an option to write out the configuration to a file named `kodo.properties`. This file, if written to, will be stored at the root of your project source directory.

### Note

To users of previous versions of this plugin, the default `kodo.properties` is no longer actively maintained by the configuration panel. Instead, you must actively propagate changes to your file by selecting the `Write to kodo.properties?` option.

## 15.4.3. Creating and building JDO projects in JBuilder

---

When right-clicking on one or more `.java` in JBuilder, you will see an option to "Create Kodo JDO Metadata". This will create a `.jdo` metadata file for all of the classes. If only one class is selected, a corresponding single class `.jdo` file is created. If multiple classes of a single package are selected, a package level `.jdo` file is created. Otherwise, a `system.jdo` is created. See **JDO Metadata Placement** for more details on metadata placement.

The JDO enhancer will be automatically run on any `.jdo` file in the project. Furthermore, `.jdo` and `.mapping` files will be copied over to the output directory, so that it will be available at runtime. This behavior can be triggered manually by compiling `.jdo` and `.mapping` files.

## 15.4.4. 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.

## 15.4.5. Editing Mapping Info from JBuilder

---

### Note

Editing mapping info requires understanding of Kodo's **MappingTool and mapping process**. Please read that section before manually editing mapping information.

To generate a `.mapping` source file, select a `.jdo` file and right click to raise the context menu. Select `Create Mapping`. You can now select a schema action that will happen in parallel with the mapping generation. For further details, see **Mapping Tool**. The plugin will then generate a `.mapping` file representing the default Kodo mapping for the metadata as a child of the `.jdo`

You can now edit this as an XML file, or edit using the plugin's GUI editor by selecting `Properties` from the `.mapping` context menu.

There are a couple more Mapping Tool actions available at the .jdo context level. Drop Mapping Info and Build Schema from Mapping will first request what schema action to take before triggering the corresponding Mapping Tool action.

## 15.4.6. 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:

- **Configure** the Kodo plugin as described above and set the Kodo license key and connection properties appropriately. If you are using HSQL (or another file-based database), you should use an absolute path in the URL to ensure that all schema operations work on the same database.
- 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.
- 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 Create Mapping option from the context menu. The plugin will prompt for a schema action. Select Refresh to create the tables needed for the tutorial. You will see Animal.mapping generated as a child of the Animal.jdo node. In addition, the file will be opened for editing in XML.
- Build the project by selecting Make Project "PetShop.jpx" from the Project menu. This will also run the JDO enhancer on Animal as well as deploying the metadata files to the output directory.
- Edit petshop.properties to match your plugin configuration, including license key and database connectivity information. Set the database URL to the same value as was set in the general JBuilder Kodo properties. If you are using HSQL (or another file-based database), you should use an absolute path in the URL to ensure that all schema operations work on the same database.
- Right-click on the PetShop.java file in the top left pane, and select Run using defaults from the menu. This will run the Pet Shop example.

The Pet Shop example allows you to create and delete pets in a database. The pets have a string type -- dog, cat, giraffe, etc. -- and a price.

The Pet Shop example code demonstrates how to put together a simple Swing example, and also how to use a Kodo-specific feature to extend the PersistenceManager class to enable Swing updates to happen at the optimal time.

## 15.5. 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 editing metadata files in the IDE, easy access to the MappingTool and enhancer, links into the build process to support enhancement, and wizards to help in the creation of JDO specific files.

### 15.5.1. 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 installations of the Kodo plugin should be disabled and uninstalled before installing this version of the plugin. Ensure that old Kodo jars, including dependent jars, have been removed.

## 15.5.2. Installing Kodo into the IDE

---

Copy all of Kodo's jars *excluding* `kodo-jdo.jar`, `kodo-workbench.jar`, `kodo-jdo-runtime.jar` 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.

### Note

On some earlier versions of SunONE Studio (notably, version 4), you should remove Kodo's version of `xercesImpl.jar` as it may cause some conflicts.

In addition, one should copy to this directory the jar for the JDBC driver used to connect to your database. Copy `kodo-jdo.jar` into the `modules` directory of your home IDE directory.

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 -> JDO -> 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 `jdo-1.0.2.jar` which you installed. To explicitly use Kodo in your project, mount Kodo's libraries in a similar fashion.

## 15.5.3. Configuring the Kodo Module

---

The plugin configuration is integrated into the IDE. In the `Tools -> Options` dialog box, Kodo's options are listed under `Options -> JDO -> Kodo Settings`. Sub-settings control your Kodo development configuration, such as driver information, as well as providing defaults for the properties wizard listed below. To use the plugin, you must enter your license key and enter the connection information for your database.

Using the Kodo Settings options pane, you can configure logging verbosity of the plugin. Valid values for this setting are (from least to most verbose):

- FATAL
- ERROR
- WARN
- DEBUG
- INFO
- TRACE

## 15.5.4. 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` sub-menu.

The Kodo `Properties` wizard will assist in the run-time configuration of Kodo, using your current plugin configuration set-

tings for the default values. The wizard will create new `.properties` files to use with your projects. Options are separated into logical groupings. You can optionally test this new configuration by pressing the test button on the lower right corner.

The JDO Metadata wizard provides a set of dialogs to walk the developer through metadata generation. Simply select the type of metadata file to generate, add the classes you want described by the metadata, and then configure each class' metadata. The resulting jdo file(s) can now be used in your project.

### 15.5.5. 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 **Mapping Tool** over classes defined 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 invalid nodes are selected. Select `Tools -> Kodo` to see the available actions.

`Mapping Tool` actions will prompt for a schema action to do in parallel with your mapping action. In addition, one can optionally redirect the schema changes to a `.schema` file.

### 15.5.6. Mapping DataObject

---

Kodo integrates mapping information as Mapping DataObjects. This will allow you to edit, view, and add source control onto the mapping information used by Kodo at runtime. Select a JDO node and select `Refresh mappings` to create new JDO node. To edit the file in XML view, select `Edit`. To open and edit the file in a complete GUI editor, select `Open`.

### 15.5.7. 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.

### 15.5.8. SunONE / NetBeans Sample

---

To build and run the sample, first follow the plugin 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's libraries into the project. To run the sample, do the following:

- **Configure** the Kodo module as described above and ensure that you have set the Kodo license key and connection properties appropriately. If you are using `HSQL` (or another file-based database), you should use an absolute path in the URL to ensure that all schema operations work on the same database.
- 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 -> Refresh mappings` from the menu. Select the `Refresh schema` action and select `OK`. By selecting `Refresh schema` action will create the tables necessary for the sample.
- Edit `petshop.properties` to match your plugin configuration, including license key and database connectivity information. Set the database URL to the same value as was set for the module configuration. If you are using `HSQL` (or another file-based

database), you should use an absolute path in the URL to ensure that all schema operations work on the same database.

- Right-click on the `PetShop` node in the explorer and select `Execute` from the menu. This will run the Pet Shop example.

The Pet Shop example allows you to create and delete pets in a database. The pets have a string type -- dog, cat, giraffe, etc. -- and a price.

The Pet Shop example code demonstrates how to put together a simple Swing example, and also how to use a Kodo-specific feature to extend the `PersistenceManager` class to enable Swing updates to happen at the optimal time.

## 15.6. Eclipse / WebSphere Studio Integration

---

*The versions that the plugin has been tested are 1.0, 2.x, and 3.0 of the Eclipse IDE, and version 4 and 5 of WebSphere Studio Application Developer. While the plugin may work on other IDEs/ versions 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, MappingTool, and building projects.

### 15.6.1. Installing the Kodo Eclipse Plugin

---

#### Note

Completely uninstall previous versions of the Kodo plugin as it may interfere with the new plugin.

First copy the directory `kodo.eclipse_2.1.0` directory from the `eclipse` directory in your Kodo installation to the `plugins` directory of the IDE base directory. For example, if you installed Kodo at `C:\Kodo` and Eclipse at `C:\Eclipse`, one would take `C:\Kodo\eclipse\kodo.eclipse_2.1.0` directory and place it such that the new directory structure would be: `C:\Eclipse\plugins\kodo.eclipse_2.1.0`.

Copy all the jars from the `lib` directory of your Kodo installation into this new directory. In addition, copy the JDBC driver of your choice into this directory as well. *Keep the driver filename in mind as you will need it later.*

In the Kodo plugin directory, modify the marked portion of `plugin.xml` to point to your JDBC driver .jar file.

Start the IDE. You should see a Kodo menu item, and Kodo listed as an available view. If not, you may need to configure your perspective to include those items. To manually configure your perspective for WSAD 4, perform the following steps. These steps are likely to work for other versions of Eclipse, but have been written with WSAD 4 in mind.

1. Click on `Perspective -> Customize...` to open the `Customize Perspective` dialog box.
2. Open up the `Views` checkbox.
3. Select `Kodo` from the list of available views (under `Java`) if it's unselected. This will add the `Kodo Java` 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.

## 15.6.2. 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, and then OK. The plugin should now be configured.

To access the full range of Kodo configuration options, or to load defaults for configuration values (which can be overridden in the IDE), you can enter the path to your `kodo.properties` file in this window.

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.

## 15.6.3. 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` files by selecting them in the explorer pane and selecting either the Enhance icon in the toolbar or selecting Kodo->Enhance from the main menu bar.

To run the **Mapping Tool** To run the SchemaTool, similarly select one or more `.jdo` file(s) and select one of the Mapping Tool icons, or the corresponding menu item. A dialog will ask you what schema operations to do in parallel with the mapping action you selected. In addition, you can redirect the schema changes to a **schema XML file**.

## 15.6.4. 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 Mapping Tool on `Animal.jdo` by selecting the file and either clicking on `Kodo->Refresh Mappings` from the menu or selecting the icon with the matching tooltip. Select refresh as the schema action to create the necessary tables for the tutorial.

Now alter `petshop.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 16. Optimization Techniques

There are numerous techniques the developer can use in order to ensure that Kodo JDO operates in the fastest and most efficient manner. Following are some guidelines. These hints contain information about what impact they will have on performance and scalability. Note that general guidelines regarding performance or scalability issues are just that -- guidelines. Depending on the particular characteristics of your application, the optimal settings may be considerably different than what is outlined below.

In the following table, each row is labeled with a list of italicized keywords. These keywords identify what characteristics the row in question may improve upon.

Many of the rows are marked with one or both of the *performance* and *scalability* labels. It is important to bear in mind the differences between performance and scalability (for the most part, we are referring to system-wide scalability, and not necessarily only scalability within a single JVM). The performance-related hints will probably improve the performance of your application for a given user load, whereas the scalability-related hints will probably increase the total number of users that your application can service. Sometimes, increasing performance will decrease scalability, and vice versa. Typically, options that reduce the amount of work done on the database server will improve scalability, whereas those that push more work onto the server (for example, flushing before queries when changes have been detected and IgnoreCache is set to false) will have a negative impact on scalability.

**Table 16.1. Optimization Techniques**

<b>Optimize database indexes</b> <i>performance, scalability</i>	<p>The default set of indexes created by Kodo JDO's mapping tool may not always be the most appropriate for your application. Using Kodo's <code>jdbc-indexed</code> metadata extension or manually manipulating indexes to include frequently-queried fields (as well as dropping indexes on rarely-queried fields) can yield significant performance benefits.</p> <p>A database must do extra work on insert, update, and delete to maintain an index. This extra work will benefit selects with WHERE clauses, which will execute much faster when the terms in the WHERE clause are appropriately indexed. So, for a read-mostly application, appropriate indexing will slow down updates (which are rare) but greatly accelerate reads. This means that the system as a whole will be faster, and also that the database will experience less load, meaning that the system will be more scalable.</p> <p>Bear in mind that over-indexing is a bad thing, both for scalability and performance, especially for applications that perform lots of inserts, updates, or deletes.</p>
<b>Use the best JDBC driver</b> <i>performance, scalability, reliability</i>	<p>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 JDO's data access and increase load on the database, reducing system performance and scalability.</p>
<b>JVM optimizations</b> <i>performance, reliability</i>	<p>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 <a href="http://java.sun.com/docs/hotspot/PerformanceFAQ.html">http://java.sun.com/docs/hotspot/PerformanceFAQ.html</a>.</p>
<b>Use the data cache</b> <i>performance, scalability</i>	<p>Using Kodo's <b>data caching and query caching</b> features (available in Kodo JDO Performance Pack and Enterprise Edition) can often result in a dramatic improvement in performance. Additionally, these caches can significantly reduce the amount of load on the database, increasing the scalability characteristics of your application.</p>
<b>Disable logging, performance tracking</b> <i>performance</i>	<p>Developer options such as verbose logging and the JDBC performance tracker can result in serious performance hits for your application. Before evaluating any Kodo JDO's performance, these options should all be disabled.</p>
<b>Set IgnoreCache to true, or set FlushBe-</b>	<p>When both the <code>javax.jdo.option.IgnoreCache</code> and <code>kodo.FlushBeforeQueries</code> properties are set to false, Kodo needs to evaluate in-memory dirty instances against the datastore values</p>

<p><b>foreQueries to true</b></p> <p><i>performance vs. scalability</i></p>	<p>that are returned from a query. This can sometimes result in Kodo needing to evaluate the entire extent of objects in order to return the correct query results, which can have drastic performance consequences. If it is appropriate for your application, configuring <code>FlushBeforeQueries</code> to automatically flush queries will ensure that this never happens. Setting <code>IgnoreCache</code> to false will result in a small performance hit even if <code>FlushBeforeQueries</code> 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.</p> <p>Note that the default setting of <code>FlushBeforeQueries</code> is <code>with-connection</code>, which means that data will be flushed only if a dedicated connection is already in use by the persistence manager. So, the default value may not be appropriate for you.</p> <p>Setting <code>IgnoreCache</code> to true will help performance, since the persistence manager cache can be ignored for queries, meaning that incremental flushing or client-side processing is not necessary. It will also improve scalability, since overall database server usage is diminished. On the other hand, setting <code>IgnoreCache</code> to false will have a negative impact on scalability, even when using automatic flushing before queries, since more operations will be performed on the database server.</p>
<p><b>Configure <code>kodo.ConnectionRetainMode</code> appropriately</b></p> <p><i>performance vs. scalability</i></p>	<p>The <code>ConnectionRetainMode</code> configuration option controls when Kodo will obtain a connection, and how long it will hold that connection. The optimal settings for this option will vary considerably depending on the particular behavior of your application. You may even benefit from using different retain modes for different parts of your application.</p> <p>The default setting of <code>on-demand</code> minimizes the amount of time that Kodo holds onto a datastore connection. This is generally the best option from a scalability standpoint, as database resources are held for a minimal amount of time. However, if your connection pool is overly small relative to the number of concurrent persistence managers that need access to the database, or if your <code>DataSource</code> is not efficient at managing its pool, then this default value could cause undesirable pool contention.</p>
<p><b>Ensure that batch updates are available</b></p> <p><i>performance, scalability</i></p>	<p>When performing bulk inserts, updates, or deletes, Kodo JDO will use batched statements. If this feature is not available in your JDBC driver, then Kodo JDO will need to issue multiple SQL statements instead of a single batch statement.</p>
<p><b>Use single-table inheritance</b></p> <p><i>performance, scalability vs. disk space</i></p>	<p>Using a single-table (flat) inheritance model is faster for most operations than a multi-table (vertical) inheritance model or horizontal mappings. If it is appropriate for your application, you should use the single-table inheritance model whenever possible.</p> <p>However, single-table inheritance will require more disk space on the database side. Disk space is relatively inexpensive, but if your object model is particularly large, this can become a factor.</p>
<p><b>High increment in the sequence factory</b></p> <p><i>performance, scalability</i></p>	<p>For applications that perform large bulk inserts, the retrieval of sequence numbers can be a bottleneck. Increasing the value of the <code>Increment</code> property of the <code>kodo.jdbc.SequenceFactory</code> plugin can reduce or eliminate this bottleneck. In some cases, implementing your own sequence factory can further optimize sequence number retrieval.</p>
<p><b>Use optimistic transactions</b></p> <p><i>performance, scalability</i></p>	<p>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.</p> <p>Optimistic transactions provide the same transactional guarantees as datastore transactions, except that you must handle a potential optimistic verification exception at the end of a transaction instead of assuming that a transaction will successfully complete. In many applications, it is unlikely that different concurrent transactions will operate on the same set of data at the same time, so optimistic verification increases the concurrency, and therefore both the performance and scalability characteristics, of the application. A common approach to handling optimistic verification exceptions is to simply present the end user with the fact that concurrent modifications happened, and require that the user redo any work.</p>
<p><b>Use query aggregates and projections</b></p>	<p>Using aggregates to compute reporting data on the database server can drastically speed up queries. Similarly, using projections when you are interested in specific object fields or relations rather than the entire object state can reduce the amount of data Kodo must transfer from the database to your applica-</p>

<i>performance, scalability</i>	tion. See <b>Chapter 11, Query</b> [54] for details.
<b>Perform nontransactional data reads outside datastore (pessimistic) transactions</b> <i>performance, scalability</i>	When using optimistic transactions, there is very little overhead involved in starting a transaction, so this does not help out very much in those situations.
<b>Always close persistence managers, extent iterators, and query results</b> <i>scalability</i>	<p>Under certain settings, these objects may be backed by resources in the database. For example, if you have configured Kodo to use scrollable cursors and lazy object instantiation by default, each query result will hold open a <code>ResultSet</code> object, which, in turn, will hold open a <code>Statement</code> object (preventing it from being re-used). Garbage collection will clean up these resources, so it is never necessary to explicitly close them, but it is always faster if it is done at the application level.</p> <p><b>Example 16.1. Explicitly Closing Resources</b></p> <pre> public void giveRaise (String jdoql, double amnt) {     PersistenceManagerFactory factory = ...;     PersistenceManager pm = factory.getPersistenceManager ();     Query query = null;     try     {         query = pm.newQuery (Employee.class, jdoql);         Collection res = (Collection) query.execute ();         for (Iterator itr = res.iterator (); itr.hasNext ();)         {             Employee emp = (Employee) itr.next ();             emp.setSalary (emp.getSalary () * (1 + amnt));         }     }     finally     {         if (query != null)             query.closeAll ();         pm.close ();     } } </pre>
<b>Optimize connection pool settings</b> <i>performance, scalability</i>	Kodo JDO's built-in connection pool's default settings may not be optimal for all applications. For applications that instantiate and close many <code>PersistenceManagers</code> (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. You may want to tune the prepared statement pool size with the connection pool size.
<b>Utilize the persistence manager cache</b> <i>performance, scalability</i>	When possible and appropriate, re-using persistence managers and setting the <b>RetainValues</b> configuration option to <code>true</code> may result in significant performance gains, since the persistence manager's built-in object cache will be used.
<b>Enable multithreaded operation only when necessary</b> <i>performance</i>	Kodo JDO respects the <code>javax.jdo.option.Multithreaded</code> option in that it does not impose synchronization overhead for applications that set this value to <code>false</code> . If your application is guaranteed to only access a given persistence manager or related objects (extent, query) from a single thread, setting this option to <code>false</code> will result in the elimination of synchronization overhead, and may result in a modest performance increase.
<b>Enable large data set handling</b> <i>performance, scalability</i>	If you execute queries that return large numbers of objects or have relations (collections or maps) that are large, and if you often only access parts of these data sets, enabling <b>large result set handling</b> where appropriate can dramatically speed up your application, since Kodo will bring the data sets into memory from the database only as necessary.
<b>Disable large data set</b>	If you have enabled scrollable result sets and on-demand loading but do you not require it, consider

<b>handling</b> <i>performance, scalability</i>	<p>disabling it again. Some JDBC drivers and databases (SQLServer for example) are much slower when used with scrolling result sets.</p>
<b>Develop a custom class indicator, use the metadata-value indicator with short symbolic constants, or do not use class indicators</b> <i>performance, scalability</i>	<p>Kodo JDO's default class indicator 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 <b>metadata-value</b> indicator and a little application-specific configuration, you could easily reduce this to a single character or 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.</p> <p>Alternately, if certain persistent classes in your application do not make use of inheritance, then you can disable the class indicator for these classes altogether.</p> <p><b>Example 16.2. Disabling the Class Indicator</b></p> <pre>&lt;jdo&gt;   &lt;package name="com.xyz"&gt;     &lt;class name="NoSubclasses"&gt;       &lt;extension vendor-name="kodo" key="jdbc-class-ind-name" value="none"/&gt;       &lt;!-- rest of class metadata --&gt;     &lt;/class&gt;   &lt;/package&gt; &lt;/jdo&gt;</pre> <p>If you use some sort of mapping of symbolic constants to subclasses, bear in mind that changes to your class structure will require a bit more care, since you must take care to maintain the extra indirection from class indicator value to actual value.</p>
<b>Use the Dynamic-SchemaFactory</b> <i>performance, validation</i>	<p>Kodo JDO's default schema factory reflects on the database schema to validate that object-relational mapping information is valid when a persistent class is first used. This can be a slow process on some databases. Though the database reflection is only performed once for each class, switching the <b>kodo.jdbc.SchemaFactory</b> configuration property to <code>dynamic</code> can reduce the warm-up time for your application. Note, however, that the dynamic schema factory does not perform any validation and cannot detect foreign key constraints (and is therefore not able to re-order statements to satisfy any non-deferrable non-nullable foreign keys).</p>
<b>Do not use XA transactions</b> <i>performance, scalability</i>	<p><b>XA transactions</b> can be orders of magnitude slower than standard transactions. Unless distributed transaction functionality is required by your application, use standard transactions.</p> <p>Recall that XA transactions are distinct from managed transactions -- managed transaction services such as that provided by EJB declarative transactions can be used both with XA and non-XA transactions. XA transactions should only be used when a given business transaction involves multiple different transactional resources (an Oracle database and an IBM transactional message queue, for example).</p>
<b>Use Sets instead of List/Collections</b> <i>performance, scalability</i>	<p>There is a small amount of extra overhead for Kodo to maintain collections where each element is not guaranteed to be unique. If your application does not require duplicates for a collection, you should always declare your fields to be of type <code>Set</code>, <code>SortedSet</code>, <code>HashSet</code>, or <code>TreeSet</code>.</p>
<b>Use JDOQL parameters instead of encoding search data in filter strings</b> <i>performance</i>	<p>If your queries depend on parameter data only known at runtime, you should use JDOQL parameters rather than dynamically building different query filters. Kodo performs aggressive caching of both query compilation data and <code>PreparedStatement</code>s, and the effectiveness of both of these caches are diminished if multiple query filters are used where a single one could have been used.</p>

**Example 16.3. Appropriate use of JDOQL parameters**

```
public Person findPerson (String firstName, String lastName)
{
    PersistenceManager pm = factory.getPersistenceManager ();
    try
    {
        // good -- the query uses parameters
        Query query = pm.newQuery (Person.class);
        query.setFilter ("firstName == :fname && lastName == :lname");
        Collection res = (Collection) query.execute (firstName, lastName);
        Iterator itr = res.iterator ();
        return (itr.hasNext ()) ? (Person) itr.next () : null;
    }
    finally
    {
        if (query != null)
            query.closeAll ();
        pm.close ();
    }
}
```

**Example 16.4. Inappropriate use of JDOQL parameters**

```
public Person findPerson (String firstName, String lastName)
{
    PersistenceManager pm = factory.getPersistenceManager ();
    Query query = null;
    try
    {
        // bad -- the query encodes parameters directly in filter
        query = pm.newQuery (Person.class);
        query.setFilter ("firstName == '" + firstName
            + "' && lastName == '" + lastName + "'");
        Collection res = (Collection) query.execute ();
        Iterator itr = res.iterator ();
        return (itr.hasNext ()) ? (Person) itr.next () : null;
    }
    finally
    {
        if (query != null)
            query.closeAll ();
        pm.close ();
    }
}
```

**Tune your fetch groups appropriately**  
*performance, scalability*

The **fetch groups** used when loading an object control how much data is eagerly loaded, and by extension, which fields must be lazily loaded at a future time. The ideal fetch group configuration loads all the data that is needed in one fetch, and no extra fields -- this minimizes both the amount of data transferred from the database, and the number of trips to the database.

If extra fields are specified in the fetch groups (in particular, large fields such as binary data, or relations to other persistence-capable objects), then network overhead (for the extra data) and database processing (for any necessary additional joins) will hurt your application's performance. If too few fields are specified in the fetch groups, then Kodo will have to make additional trips to the database to load additional fields as necessary.

**Use eager fetching**  
*performance, scalability*

Using **eager fetching** when loading subclass data or traversing relations for each instance in a large collection of results can be sped up by considerably by employing eager fetching (available in Kodo JDO Performance Pack).

---

## **Part VI. Kodo JDO Examples**

---

---

# Table of Contents

- 1. Kodo Sample Code ..... 369
  - 1.1. Using Application Identity ..... 369
  - 1.2. Using JDO with Java Server Pages (jsp) ..... 369
  - 1.3. Custom Proxies ..... 369
  - 1.4. JDO Enterprise Java Beans Facade ..... 369
  - 1.5. Customizing Logging ..... 369
  - 1.6. Custom Sequence Factory ..... 369
  - 1.7. Horizontal Mappings ..... 369
  - 1.8. Using Externalization to Persist Second Class Objects ..... 370
  - 1.9. Using Persistent Classes Without Enhancement ..... 370
  - 1.10. XDoclet Integration ..... 370
  - 1.11. Custom Mappings ..... 370
  - 1.12. Example of full-text searching in JDO ..... 371
  - 1.13. JMX Management ..... 371
  - 1.14. XML Store Manager ..... 372
  - 1.15. Sample Human Resources Model ..... 373
  - 1.16. Sample School Schedule Model ..... 373

---

# Chapter 1. Kodo Sample Code

The Kodo distribution comes with a number of examples that illustrate the usage of various features.

## 1.1. Using Application Identity

---

The files for this sample are located in the `samples/appid` directory of the Kodo installation. This sample shows how to use application identity. The `GovernmentForm` class uses a static inner class as its application identity class. The `Main` class is a simple JDO application that allows you to create and manipulate persistent `GovernmentForm` instances.

## 1.2. Using JDO with Java Server Pages (jsp)

---

The files for this sample are located in the `samples/jsp` directory of the Kodo installation. This sample creates a simple Pet-shop web application to demonstrate using JDO with JSPs. Run `ant` on the included `build.xml` file and follow the instructions `ant` displays.

## 1.3. Custom Proxies

---

The files for this sample are located in the `samples/proxies` directory of the Kodo installation. This sample demonstrates custom second class object proxies for mutable persistent fields not directly supported by JDO.

- `CustomSet` is a custom extension of `java.util.Set` with some extra methods.
- `CustomStringContainer` is a mutable object that holds an internal string. We use this object as a second class object, not a first class object.

The `CustomProxies` class is a persistence-capable class that uses each of the custom proxies above. The `CustomProxiesMain` and is a simple driver program for showing that the custom proxies work.

## 1.4. JDO Enterprise Java Beans Facade

---

The files for this sample are located in the `samples/ejb` directory of the Kodo installation. This sample demonstrates how to use JDO with EJBs. It includes both JDO-backed BMP entity beans and JDO-using session beans.

## 1.5. Customizing Logging

---

The files for this sample are located in the `samples/logging` directory of the Kodo installation. This sample shows how to plug a custom logging strategy into Kodo. The `Main` class implements custom logging and has a `main` method demonstrating how to set your custom log as the system default.

## 1.6. Custom Sequence Factory

---

The files for this sample are located in the `samples/seqfactory` directory of the Kodo installation. This sample shows how to define a custom sequence factory and use it for certain classes.

## 1.7. Horizontal Mappings

---

This sample demonstrates how to use horizontal mappings to implement the common pattern of individual tables that track the

creation data and last modification date. This is accomplished through the `LastModified` superclass that contains a `lastModificationDate` and `creationDate` Date field that is automatically updated from the `jdoPreStore` method.

The `package.mapping` file specifies that the `LastModified` class use a horizontal mapping, which means that each of the fields will then be declared in the subclasses: `Widget`, `WidgetOrder`, `WidgetOrderItem`.

## 1.8. Using Externalization to Persist Second Class Objects

---

The files for this sample are located in the `samples/externalization` directory of the Kodo installation. This sample demonstrates how to persist field types that aren't directly supported by JDO using Kodo's externalization framework.

The `ExternalizationFields` class is a persistence-capable class with fields of various types, none of which are recognized by JDO. The JDO metadata for `ExternalizationFields` uses Kodo's "externalizer" and "factory" metadata extensions to name methods that can be used to transform each field into a supported type, and then reconstruct it from its external form. Even complex external forms are supported; the `ExternalizationFields.pair` field externalizes to a list of persistence-capable objects.

The `ExternalizationFieldsMain` class is a driver to demonstrate that `ExternalizationFields` instances persist correctly.

## 1.9. Using Persistent Classes Without Enhancement

---

The files for this sample are located in the `samples/noenhancement` directory of the Kodo installation. The classes in this sample demonstrate how to implement the `javax.jdo.PersistenceCapable` interface in source code, without enhancement.

## 1.10. XDoclet Integration

---

The files for this sample are located in the `samples/xdoclet` directory of the Kodo installation. This sample demonstrates how to use XDoclet comment tags to create JDO metadata. The `Main` class is a persistent type with appropriate comment tags. The `build.xml` file invokes XDoclet to create a JDO metadata file from the commented source.

## 1.11. Custom Mappings

---

The files for this sample are located in the `samples/ormapping` directory of the Kodo installation. This sample demonstrates custom field and class mappings.

- `IsMaleMapping` is a custom field mapping that transforms a boolean field into 'M' or 'F' characters in the database. It shows how to create simple transformation mappings.
- `SQLDateMapping` is a custom field mapping for `java.sql.Date`, which is not supported by JDO. It shows how to create mappings for standard JDBC types that are not covered by the JDO spec.
- `XMLMapping` is a custom field mapping that simulates a field that maps to a non-standard column type, and that may require non-standard operations to store and retrieve data.
- `PointMapping` is a custom field mapping for `java.awt.Point`. It demonstrates how to create a multi-column custom field mapping for complex data.
- `StoredProcClassMapping` is a custom class mapping that simulates using stored procedures to access persistent data.

The `CustomFields` class is a persistence-capable class that uses each of the custom field mappings above. The `StoredProc` class is a persistence-capable class that uses the `StoredProcClassMapping`. The `CustomFieldsMain` and `StoredProcMain` classes are

simple driver programs for each of these types.

Note that creating custom class mappings requires a license for the Enterprise Edition. Custom field mappings work with the Standard Edition.

Also, make sure to browse the "externalization" sample directory. Kodo includes an externalization feature that can be used to persist many unsupported field types without having to create a custom field mapping.

## 1.12. Example of full-text searching in JDO

---

The files for this sample are located in the `samples/textindex` directory of the Kodo installation. This sample demonstrates how full-text indexing might be implemented in JDO. Most relational databases cannot optimize contains queries for large text fields, meaning that any substring query will result in a full table scan (which can be extremely slow for tables with many rows).

The `AbstractIndexable` class implements `javax.jdo.InstanceCallbacks` which will cause the textual content of the implementing persistent class to be split into individual "word" tokens, and stored in a related table. Since this happens whenever an instance of the class is stored, the index is always up to date. The `Indexer` class is a utility that assists in building queries that act on the indexed field.

The `TextIndexMain` class is a driver to demonstrate a simple text indexing application.

## 1.13. JMX Management

---

This sample shows how to use the Kodo JMX Management technology preview.

In order to see an example of remote management:

- Ensure that the Kodo distribution root directory is in your CLASSPATH
- Ensure that an appropriate `kodo.properties` is in a directory in your CLASSPATH or in the root level of a jar in your CLASSPATH
- Ensure that your CLASSPATH has all of the jars distributed with Kodo
- **`javac *.java`**
- **`jdoc package.jdo`**
- **`mappingtool -action refresh package.jdo`**
- **`java samples.management.SeedDatabase`**
- Add the following line to your `kodo.properties` file:  
**`kodo.ManagementConfiguration: remote-mgmt`**
- In one window, run: **`java samples.management.ManagementSampleRemote`**
- Within a few seconds, in a second window, run: **`remotejmxtool`**

The Kodo `ProfilingAgent` also has a JMX MBean. You can see an example of remote profiling if you turn on the `ProfilingAgent` by making the following setting in your `kodo.properties` file:

- **`kodo.ManagementConfiguration: remote-mgmt-prof`**

The Kodo ProfilingAgent also has a local mode. You can see an example of local profiling if you turn on the ProfilingAgent by making the following setting in your `kodo.properties` file:

- **`kodo.ManagementConfiguration: profiling-gui`**

A fetch group `g` has been pre-configured in the `package.jdo` file. You can turn on the usage of that fetch group in the example by adding the following setting to your `kodo.properties` file:

- **`kodo.FetchGroups: g`**

Note that the Kodo PersistenceManagerFactory DataCache also has a JMX MBean. You can see an example of DataCache management if you turn on the data cache by adding the following settings to your `kodo.properties` file:

- **`kodo.DataCache: true`**
- **`kodo.RemoteCommitProvider: sjvm`**

An example of how to use the TimeWatch can be found in `QueryThread.java`. A TimeWatch allows for monitoring of named code blocks.

---

## 1.14. XML Store Manager

---

This sample shows how to use the Kodo XML Store Manager `AbstractStoreManager` implementation.

For more information on implementing your own custom store manager, please see the XML Store Manager Javadoc (package `kodo.xmlstore`), and the source code in `src/kodo/xmlstore` under your Kodo installation.

### Note

The `AbstractStoreManager` requires a Kodo JDO Enterprise Edition license in order to function. See [Chapter 13, Enterprise Edition \[321\]](#) for details about the Enterprise Edition.

In order to see an example of the XML Store Manager:

- Ensure that the Kodo distribution root directory is in your `CLASSPATH`. This is done automatically if you set up your environment as described in the README.
- Ensure that the sample `xmlstore.properties` is in a directory in your `CLASSPATH` or in the root level of a jar in your `CLASSPATH`. Enter your Kodo license key into the sample `xmlstore.properties` as the value of the **`kodo.LicenseKey`** property.
- Ensure that your `CLASSPATH` has all of the jars distributed with Kodo. This is done automatically if you set up your environment as described in the README.
- **`javac *.java`**
- **`jdoc package.jdo`**
- Note that there is no need to run **`mappingtool`** for custom store managers.
- **`java samples.xmlstore.SeedDatabase`**

- Note the XML data in the selected data directory as specified by your `javax.jdo.option.ConnectionURL`.

## 1.15. Sample Human Resources Model

---

The files for this sample are located in the `samples/models/humres` directory of the Kodo installation. This sample demonstrates the mapping of an example "Human Resources" schema. The following concepts are illustrated in this sample:

- Mixed Application Identity and Datastore Identity ([Section 4.5, “JDO Identity” \[21\]](#))
- Named Query execution ([Section 11.10, “Named Queries” \[72\]](#))
- Value Mappings ([Section 7.9.1, “Value Mapping” \[236\]](#))
- One to One Mappings ([Section 7.9.5, “One-to-One Mapping” \[242\]](#))
- One to Many Mappings (with and without inverses) ([Section 7.9.11, “One-to-Many Mapping” \[256\]](#))

## 1.16. Sample School Schedule Model

---

The files for this sample are located in the `samples/models/school` directory of the Kodo installation. This sample demonstrates different inheritance strategies for the same object model. The following concepts are illustrated in this sample:

- Flat Inheritance Mappings ([Section 7.6.2, “Flat Inheritance Mapping” \[220\]](#))
- State Image Version Indicator ([Section 7.7.3, “State Image Indicator” \[231\]](#))
- Vertical Inheritance Mappings ([Section 7.6.3, “Vertical Inheritance Mapping” \[222\]](#))
- Horizontal Inheritance Mappings ([Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#))
- Value Mappings ([Section 7.9.1, “Value Mapping” \[236\]](#))
- Many to Many Mappings ([Section 7.9.10, “Many-to-Many Mapping” \[253\]](#))

The `samples/models/school` directory contains the following interfaces that represent the abstract data model:

- Address
- HomeAddress (extends Address)
- WorkAddress (extends Address)
- Person
- Parent (extends Person)
- Student (extends Person)
- Employee (extends Person)
- Teacher (extends Employee)
- Admin (extends Employee)

- Staff (extends Employee)
- Course
- ScheduledClass

Each of the `flat`, `vertical`, and `horizontal` subdirectories contain implementations of the object model that are identical except for name, and that their inheritance mapping in the `package .jdo` file uses the type of the directory's corresponding name.

---

# **Part VII. Kodo Development Workbench Guide**

---

---

# Table of Contents

1. Introduction to the Kodo Development Workbench .....	ccclxxviii
1.1. Kodo Development Workbench Requirements .....	ccclxxviii
1. Running and Configuring Kodo Development Workbench .....	379
1.1. Starting Kodo Development Workbench From the Command Line .....	379
1.2. Configuring Kodo Development Workbench .....	379
2. Getting Started with Kodo Development Workbench .....	380
2.1. Beginning The Kodo Development Workbench Tutorial .....	380
2.2. Getting Familiar with Kodo Development Workbench .....	380
2.3. The MetaData Explorer .....	380
2.3.1. MetaData Explorer Basics .....	381
2.3.2. Creating JDO MetaData .....	381
2.3.3. Using The MetaData Explorer .....	381
2.3.4. Editing MetaData .....	382
2.3.5. Additional MetaData Explorer Actions .....	382
2.4. The Schema Explorer .....	382
2.5. Kodo Development Workbench Logging .....	384
2.6. The Details Pane .....	384
2.7. The Editor .....	384
2.7.1. Editor Overview .....	384
2.7.2. Using The Visualization Editor .....	384
2.7.3. Editing Visualization Mappings .....	386
2.7.4. Mapping Fields .....	387
2.7.5. Completing Visualization Changes .....	387
2.7.6. JDOQLEditor .....	387
2.8. Running The Tutorial .....	389
3. Root MetaData Actions .....	390
3.1. Mount JDO File .....	390
3.2. Unmount Files... ..	390
3.3. Create MetaData .....	390
3.4. Import Mapping Info .....	390
4. MetaData Actions .....	391
4.1. Enhance .....	391
4.2. Edit MetaData .....	391
4.3. Add - Recreate Mapping Info .....	391
4.4. Export Mapping Info .....	391
4.5. Drop Mapping Info .....	391
4.6. Remove MetaData .....	391
4.7. Build Schema From Mapping .....	391
4.8. Visualize Mapping .....	391
5. Root Schema Actions .....	394
5.1. Run SchemaTool .....	394
5.2. Refresh Schema From DB .....	394
5.3. Create DB Script .....	394
5.4. Create Change Script .....	394
5.5. Reverse Map Schema .....	395
5.6. Import .schema file .....	395
5.7. Export to .schema file .....	395
6. Schema Actions .....	396
6.1. Drop Schema Object .....	396
6.2. Edit Table .....	396
6.3. Add .....	396
7. The Editors .....	397
7.1. MetaData Editor .....	397

7.2. The Mapping Editor .....	397
7.3. The Visualization Editor .....	398
7.4. The Table Editor .....	398
7.5. The Foreign Keys Editor .....	398
8. JDOQL Editor .....	399
8.1. Query Validator .....	399
8.2. Candidate Class Editor .....	400
8.3. Filter Editor .....	400
8.4. Additional Query Component Editors .....	400
8.4.1. Ordering Editor .....	400
8.4.2. Parameters Editor .....	400
8.4.3. Imports Editor .....	401
8.4.4. Variables Editor .....	401
8.4.5. Aggregates and Projections Editor .....	401
8.4.6. Fetch Configuration Editor .....	401
8.5. Execute Query .....	401
8.6. Show SQL .....	401
8.7. Clear Query .....	401
8.8. Show Java .....	401
8.9. Save Query .....	402
8.10. Load Query .....	402
8.11. Recent Queries .....	402
8.12. Results Browser .....	402

---

# Introduction to the Kodo Development Workbench

Kodo comes with a standalone GUI environment to visualize, edit, and maintain a JDO application throughout the development lifecycle. By providing seamless access to all of Kodo's major development tools, including the Reverse Mapping Tool and Schema Tool, Kodo Development Workbench can accelerate and simplify using all of Kodo's advanced functionality.

## 1.1. Kodo Development Workbench Requirements

---

This tutorial requires that JDK 1.3.1 or greater be installed on your computer (although JDK 1.4.2 or higher is recommended), and that your classpath and environment are set up as outlined in the initial setup instructions contained in [README.txt](#).

In addition, this guide assumes a familiarity with Kodo's **MetaData** and **Mapping** system, especially as it refers to the Mapping-Tool and MappingFactory.

---

# Chapter 1. Running and Configuring Kodo Development Workbench

This chapter will demonstrate starting and configuring Kodo Workbench, including command line options and configuring the Kodo JDO core of Kodo Workbench.

## 1.1. Starting Kodo Development Workbench From the Command Line

---

Kodo Workbench can be started by using the `kodoworkbench` shell script or by directly running the Java class, `kodo.jdbc.ide.meta.KodoWorkbench`. On Windows, you can also start the Workbench by double-clicking on the `workbench.exe` executable.

### Note

Before running the Kodo Workbench from the command line, please ensure that your environment is set up correctly as described at [Chapter 2, SolarMetric Kodo JDO Installation \[4\]](#).

Below is a listing of configurable settings for Kodo Workbench. While all the settings are optional, specifying as many of these as possible will ensure that Kodo Workbench remembers your settings and behaves as you would prefer.

- `-storage/-s /path/to/storage`: Configures where Kodo Workbench will store its persistent state. Kodo Workbench stores data in files with the ".storage" extension.
- `-directory/-d /path/to/directory`: This option configures the default directory for source files and output for generated files. A good setting would be to set it at the base of your source tree. This will default to `System.getProperty("user.home")/solarmetric`.
- `-ui ui.classname`: Optional parameter to control look and feel of Kodo Workbench. Takes a class name of a `javax.swing.LookAndFeel` implementation. Run `com.solarmetric.ide.util.IdeUtils` to get a listing of available `LookAndFeel` classes available on your platform. Note while some may be listed, they may not be available at runtime, namely Windows look and feel on non-Windows operating systems.
- `-properties/-p /path/to/kodo.properties`: Optional parameter to explicitly use a specific configuration file instead of **loading and configuring** from Kodo Workbench itself.

## 1.2. Configuring Kodo Development Workbench

---

Upon starting Kodo Workbench, a welcome dialog will appear and allow you to enter your license key, as well as the working directory for the workbench. By default, the workbench will be configured to use the pure-Java Hypersonic database, which allows you to experiment with Kodo without having to set up your database. You can always change these settings using the `Pereferences` and `Kodo Configuration` menu items.

Note that while Kodo Workbench will remember your file settings, it will not attempt to remember the contents of those files. This means that if you change the file, on future restarts, Kodo Workbench will follow the changed settings of that file.

---

# Chapter 2. Getting Started with Kodo Development Workbench

This chapter will familiarize you with Kodo Workbench's gui environment by leading you through a brief tutorial.

## 2.1. Beginning The Kodo Development Workbench Tutorial

---

This tutorial begins with compiling the source files included in the Kodo distribution. The source classes are based on modeling a simple class for a airplane ticketing application, `Passenger`. Traverse to the `samples/ide` directory and inspect the source code. Then compile the source code:

```
.../samples/ide> javac *.java
```

Now edit `idetutorial.properties` to include your license key for the workbench to use. In addition, you may want to specify an absolute file URL for the sample database to use:

```
javax.jdo.option.ConnectionURL:jdbc:hsqldb:your_kodo_directory/samples/ide/ide-db
```

Now that we have generated our classes and configured our properties, we are ready to start Kodo Workbench. If Kodo Workbench is already running, stop it so that we can configure the storage and base dirs for this tutorial as well as pick up the new classes we have just compiled. We'll restart Kodo Workbench with some temporary folders to isolate the tutorial:

```
.../samples/ide> kodoworkbench -storage tutorialstorage -properties idetutorial.properties
```

Now you that you have started Kodo Workbench, we can begin using some of Kodo Workbench's features.

## 2.2. Getting Familiar with Kodo Development Workbench

---

When you have started Kodo Workbench, the application will have all of its major panes open. Oriented to the left is the **Explorer** pane. The **Explorer** is actually two components, the **MetaData Explorer** and the **Schema Explorer**, separated by a tab system. Towards the bottom is the **Log** pane. And to the left of the Log pane should be the **Details** pane. And the majority of the screen should be taken up by the **Editor** pane. Each pane can be resized, expanded, and collapsed within each split pane. Most of your work will happen within these four panes.

On the bottom lies the **status bar**. Messages about current tasks Kodo Workbench is working on will be printed there in red. Along the top is the **menu bar**. Most of the options there will be explained in detail later. The option to exit Kodo Workbench is located under the **File** menu under **Exit**, which of course you can select at any time.

## 2.3. The MetaData Explorer

---

## 2.3.1. MetaData Explorer Basics

---

The `MetaData` tab in the *Explorer* pane contains the *MetaData Explorer*. This component represents the currently mounted JDO metadata files in the system, and in turn, the persistent classes Kodo Workbench knows about. On this component, context menus along the metadata tree hierarchy will provide access points for all metadata related actions such as running the enhancer, opening metadata to edit, and mounting and unmounting metadata files. These actions are mirrored in the `MetaData` menu item.

Individual class nodes from the explorer can be dragged and dropped into class text fields, such as assigning a persistence capable superclass or the element type for a collection field.

## 2.3.2. Creating JDO MetaData

---

First let's create a simple metadata file to give our system a single class to work with. Right click on `MetaData` and you will be presented with a context menu. This context menu holds **Root MetaData Actions**. These actions are those that apply to the entire metadata repository (vs. individual classes and their corresponding metadata).

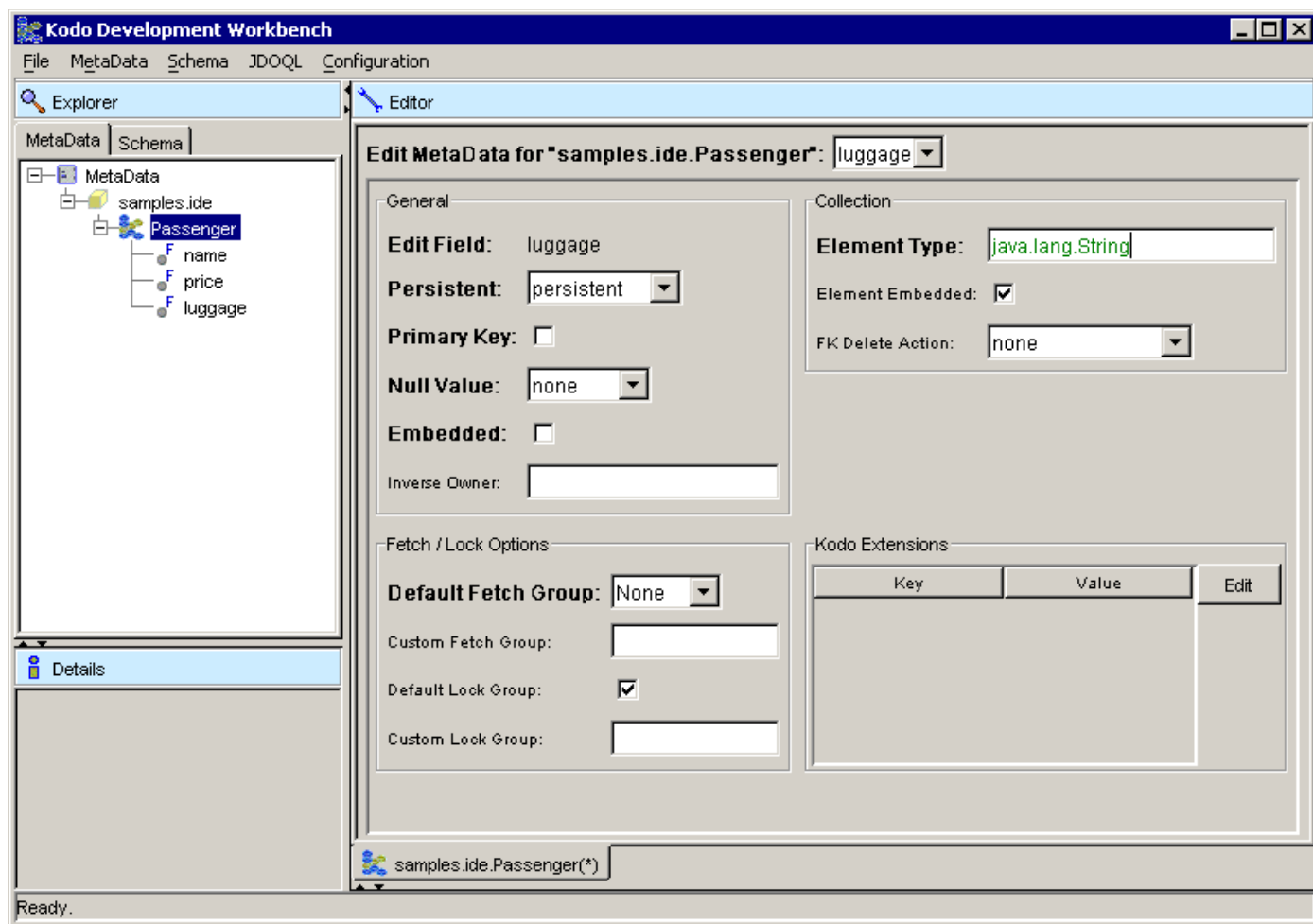
Select **Create MetaData**. This will initiate a wizard for creating new JDO metadata. Select `Class - level .jdo` and then next. The second screen of the wizard will allow you to choose classes to create metadata for. Enter `samples.ide.Passenger` and press Add. Once you have selected the class, you can press Finish.

## 2.3.3. Using The MetaData Explorer

---

You have created metadata for the `Passenger` class. You should see `Passenger` node listed once you expand the `samples.ide` node in Kodo Workbench's explorer. This metadata file is now also persistently part of Kodo Workbench. If you were to exit Kodo Workbench and return, Kodo Workbench would automatically remount the file on startup. To unmount the file later, select the **Unmount Files** Root MetaData Action.

By right clicking on the class, a new set of actions are now available. This popup menu contains *MetaData Actions*. These actions represent actions that can be done to one or more selected classes. *MetaData Actions* are where we can open **editors** for JDO metadata and Kodo mapping information. We could also remove the `Passenger` metadata from both the metadata repository and filesystem by selecting `Remove MetaData`. There are a number of *MetaData Actions* to try and use, all of which will be detailed further in a later section.



### 2.3.4. Editing MetaData

For now, open the metadata editor for `Passenger` by right clicking on the node and select **Edit MetaData**. The editor includes access to all the major JDO metadata attributes, as well as integrating components that control commonly used Kodo extensions.

Select the `luggage` field from the top drop down. Now the field-level editor will be the focus of the editor. We want this field to hold a collection of Strings, so we'll be specifying the element type for this `java.util.Set` field. Enter `java.lang.String` in the *Element Type* text field. Note that the tab representing the editor is now marked with a (\*) indicating that the editor needs to be saved. Save your changes now by selecting **Save** from the **File** menu.

### 2.3.5. Additional MetaData Explorer Actions

Now that our metadata is now tailored the way we want, we can now enhance our class. Re-select `Passenger` and select **Enhance**. This will trigger the enhancer to act on our new persistent class. You should see messages in the **Tool** category.

We'll now create mappings for the class. Having selected `Passenger`, select **Create Default Mapping**. This will cause Kodo to create some default **mapping information** as well as edit Kodo Workbench's schema which we will go into detail in the next section.

## 2.4. The Schema Explorer

The *Schema Explorer* represents the other major component of the *Explorer*. This represents a virtual **schema** that Kodo Work-

bench maintains, regardless of your database configuration. This allows you to edit, drop, and otherwise manipulate your virtual schema without long term effects on your database schema. You can drag and drop schema objects from the explorer into appropriate places into the mapping info editor, such as foreign key tables, column selections, and table text fields.

Having run `Create Default Mapping`, Kodo has built us a default schema in addition to writing mapping information into the **MappingFactory** (a `Passenger.mapping` file by default). Like the *MetaData Explorer*, the *Schema Explorer* also has two levels of actions, *Root Schema Actions* and general *Schema Actions*.

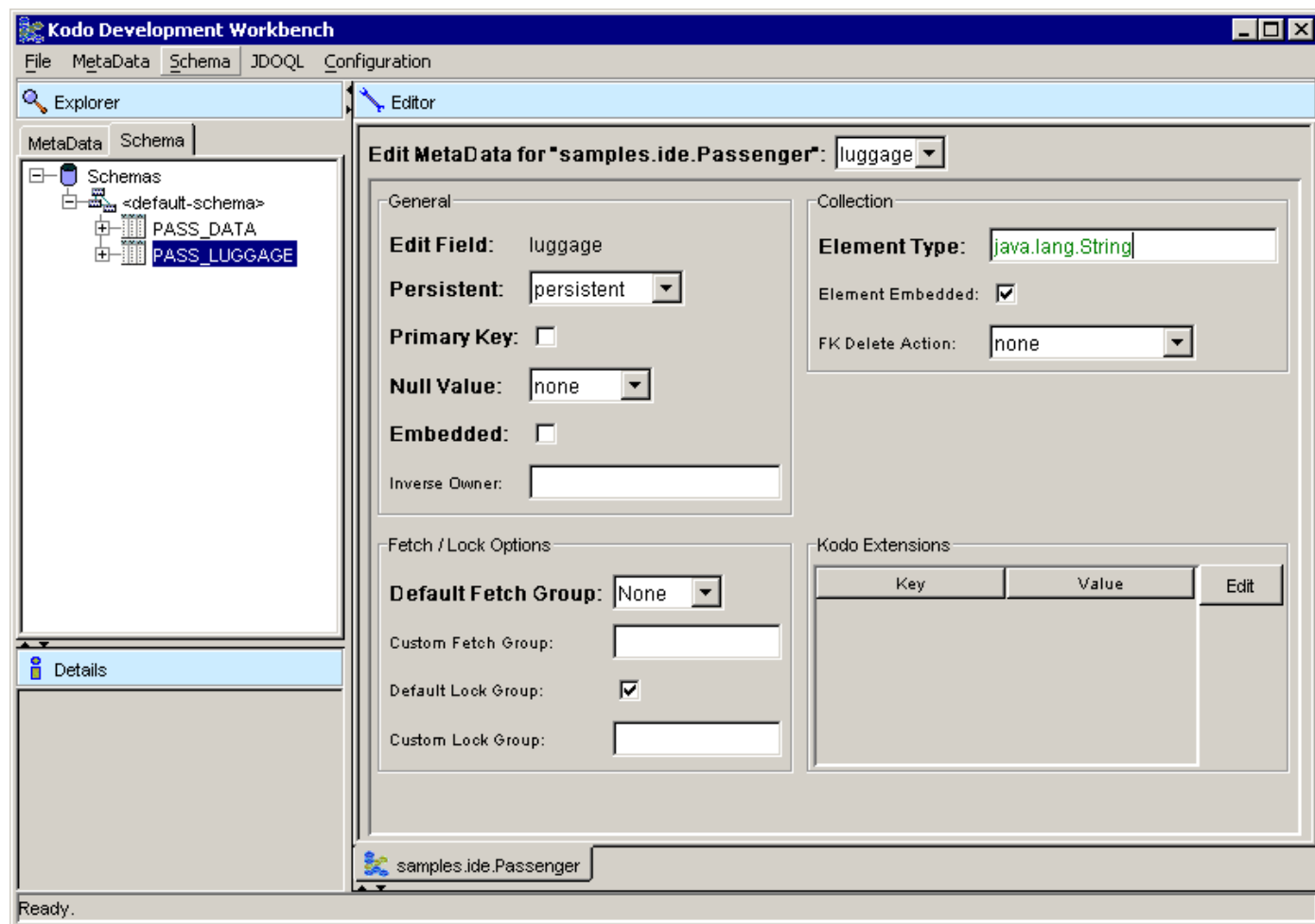
**Root Schema Actions** allow you to synchronize Kodo Workbench's schema with the actual database schema, generate database change scripts, and other database wide maintenance tasks. **Schema Actions** are actions that are sensitive to what node they are on. For example, at the schema level or lower, there is the option to drop the object from the schema. You can edit tables, add new schema objects, and more.

Continuing our tutorial, we want to use a pre-created sample database to map our newly persistent classes to. Select the *Schemas* node, and select `Refresh Schema From DB`. We'll see Kodo's auto-generated tables replaced by two new ones, `PASS_DATA` and `PASS_LUGGAGE`.

## Note

Important to note is that the virtual schema maintained by Kodo Workbench is treated like a source file. In other words, changes have to be explicitly saved back to the persistent store of Kodo Workbench. This is accomplished by selecting `Save` from the *Schema* menu.

We recommend saving your Schemas at this point in the tutorial in case you want to leave Kodo Workbench later in the process.



## 2.5. Kodo Development Workbench Logging

---

The *Log* pane is actually an implementation of `com.solarmetric.log.LogFactory` (see the **Logging** section for more details) that will log all messages and then echo the messages appropriately to your current logging configuration. We should see messages in the panels already from enhancing and adding mapping info to our system. Feel free to inspect these and adjust the logging to your needs. In addition, you may also inspect your own logging configuration to ensure that messages are being properly echoed to your own configuration.

Messages get logged here row by row sorted by date. The left column indicates the logging channel that is being written to. If you would like to see the complete text of a message, just double click on it to expand it into its own dialog.

## 2.6. The Details Pane

---

The Details pane may provide additional information about the current **editor** you are using. Primarily, this provides a toolbox for the **Visualization** tool. We'll see this pane in action a little bit later.

## 2.7. The Editor

---

### 2.7.1. Editor Overview

---

The *Editor* pane provides the area in which the majority of editing of Kodo objects occurs. Each edited object is represented on a tabbed pane. Each edited object has a text and icon representation on its tab. The tab will change to reflect changes by adding ( \* ) to the tab name.

Each tab can be saved independently using the `File` menu on the main menu bar if it has been changed. You can also close the active editor by selecting `close` from the menu bar.

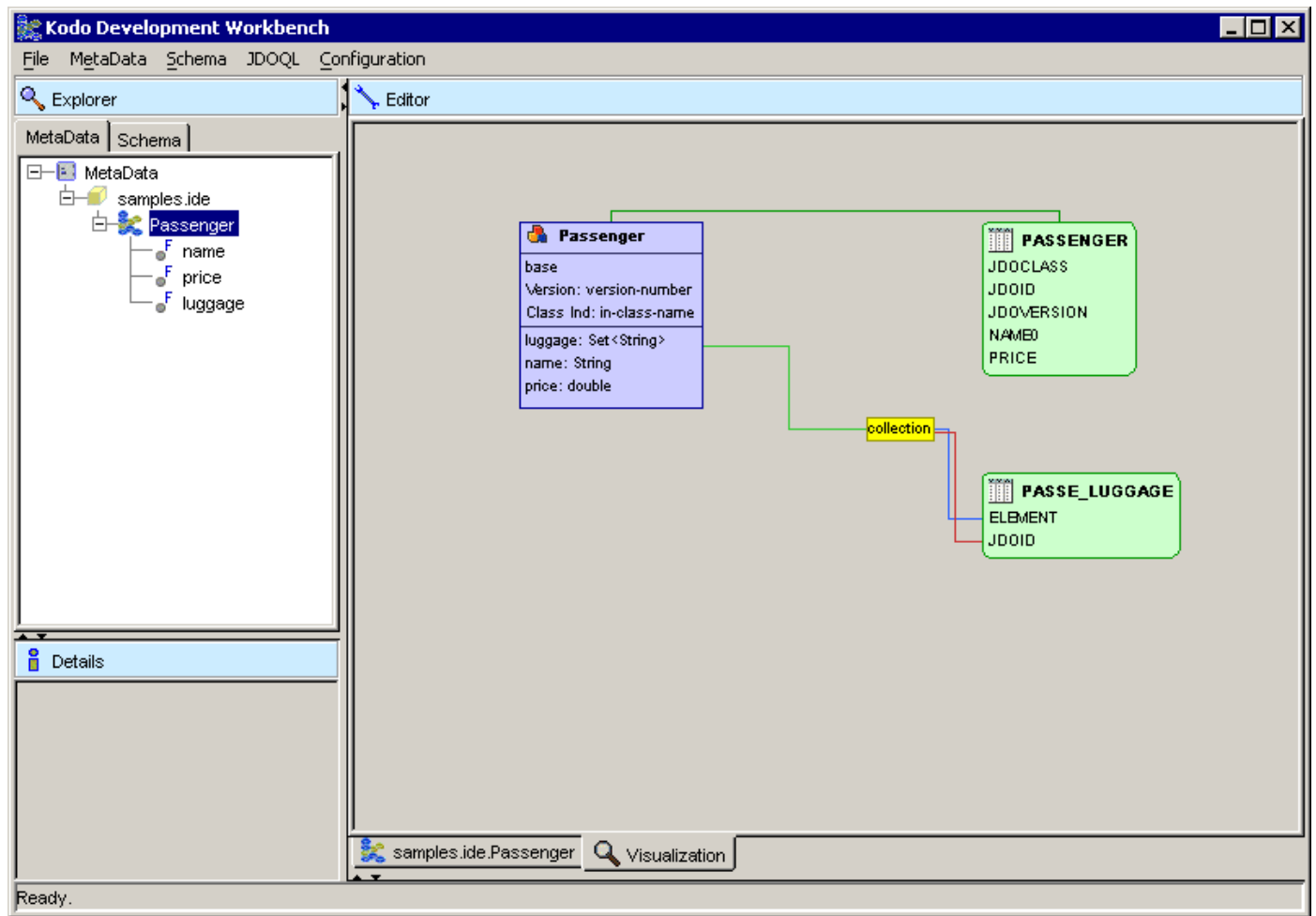
We'll use the *Visualization Editor* to illustrate how to use editors in Kodo Development Workbench. We will map our tutorial class to an existing schema, allowing our application to run without any schema changes.

### 2.7.2. Using The Visualization Editor

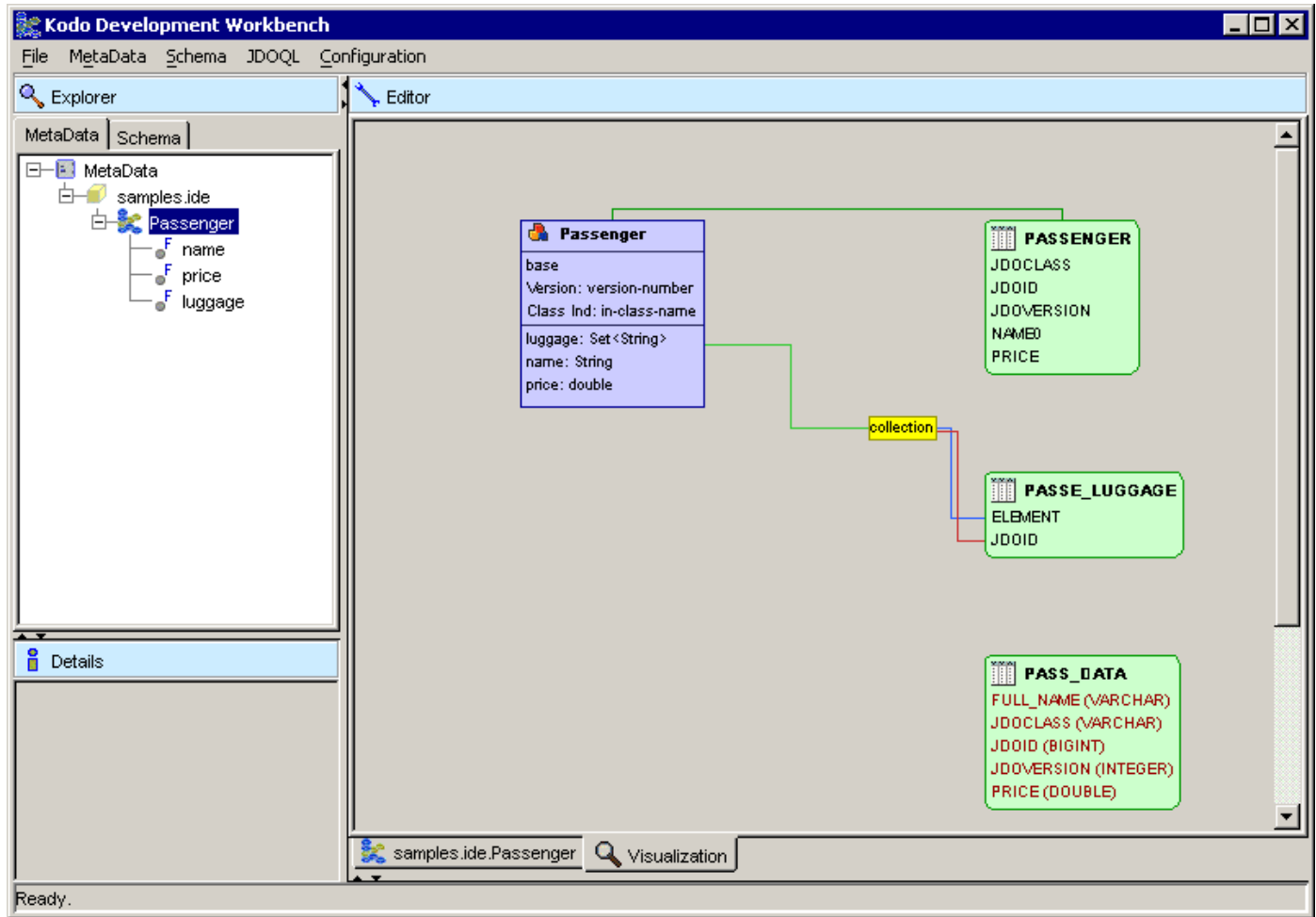
---

We'll now use **Visualization** editor, a special editor to view the interaction of classes, mappings and schema. Right-click on the `Passenger` node and then select `Visualize Mapping`.

We can see the **mapping** information that Kodo would assume at runtime that the actual schema looked like. Note that Kodo currently expects the initial auto-generated tables names we had before. We will now update this to match our pre-created schema.



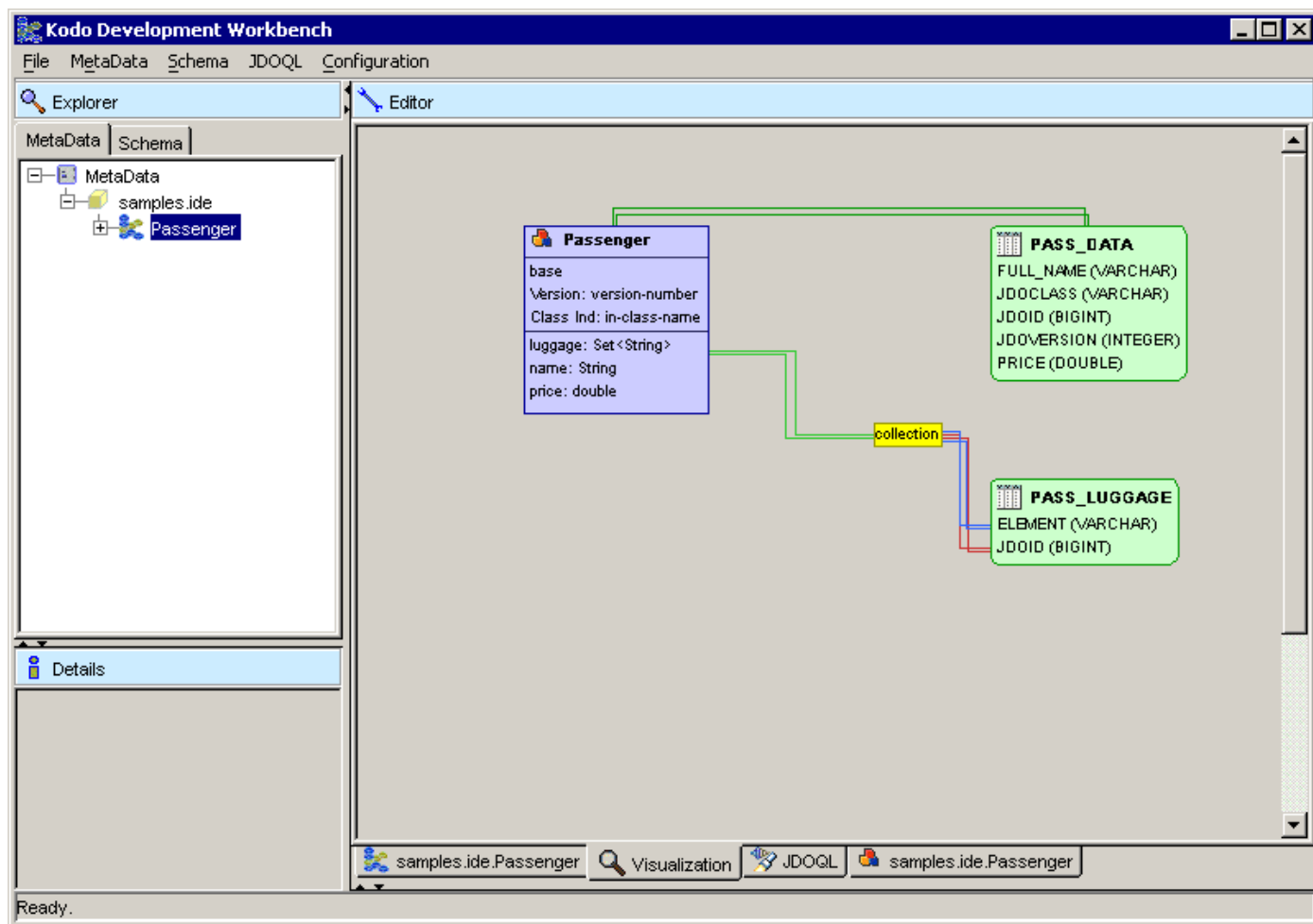
Right click on an empty area of the visualization component and select Add / Import Tables. Select the import all option, and select ok. We now can see the Workbench schema on our graph.



### 2.7.3. Editing Visualization Mappings

We will now relink the class and fields to these new tables. Go back to the Visualization editor and re-select the Passenger class node. Select the class mapping by selecting the class name on that node. You'll see a variety of colored buttons now presented to you. Select the green table button. By moving our cursor back over the graph, we can see a dashed line indicating we are attempting to link the class to a new table. Select the PASS\_DATA table.

You can see the old unused table has now disappeared and that Passenger is now linked to the PASS\_DATA table. The simple fields such as price have now been mapped to the PASS\_DATA table. However, we still need to map the name field to an existing column.



## 2.7.4. Mapping Fields

Select the name field on the class node. You can now see a different set of attributes to use in mapping the field in the Details pane. Select the blue column button. You can then select a column in PASS\_DATA. Select FULL\_NAME. Once you have done so, you can now see that the name column has been linked to a valid column, and its old column removed.

Similarly, select the yellow box marked collection. This yellow box indicates that the luggage field mapping stores the data in another table, using the collection field mapping. By selecting the node, one can see the field name in the detail pane, and the green line back to the owning class is now bold. Select again the green table button. This time we'll select PASS\_LUGGAGE.

## 2.7.5. Completing Visualization Changes

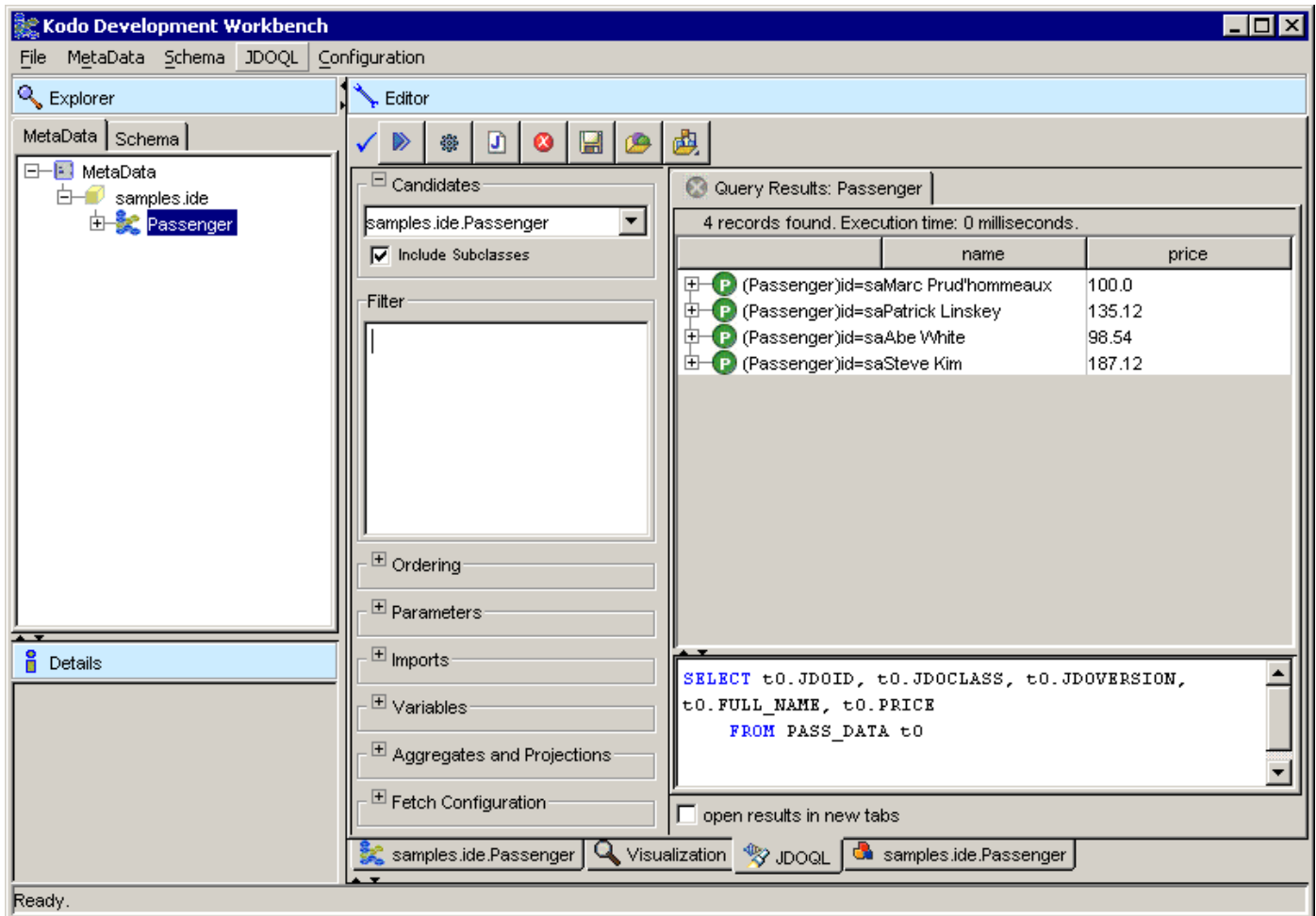
Congratulations, you have now mapped our class to two new tables. We will now save our changes so we can see the new mapping in use. Note that the visualization editor is now ready to be saved (marked with "\*"). Save the editor and we're now done editing our mappings.

The **Visualize** section of the MetaData Actions section will contain more details on how the Visualizer works.

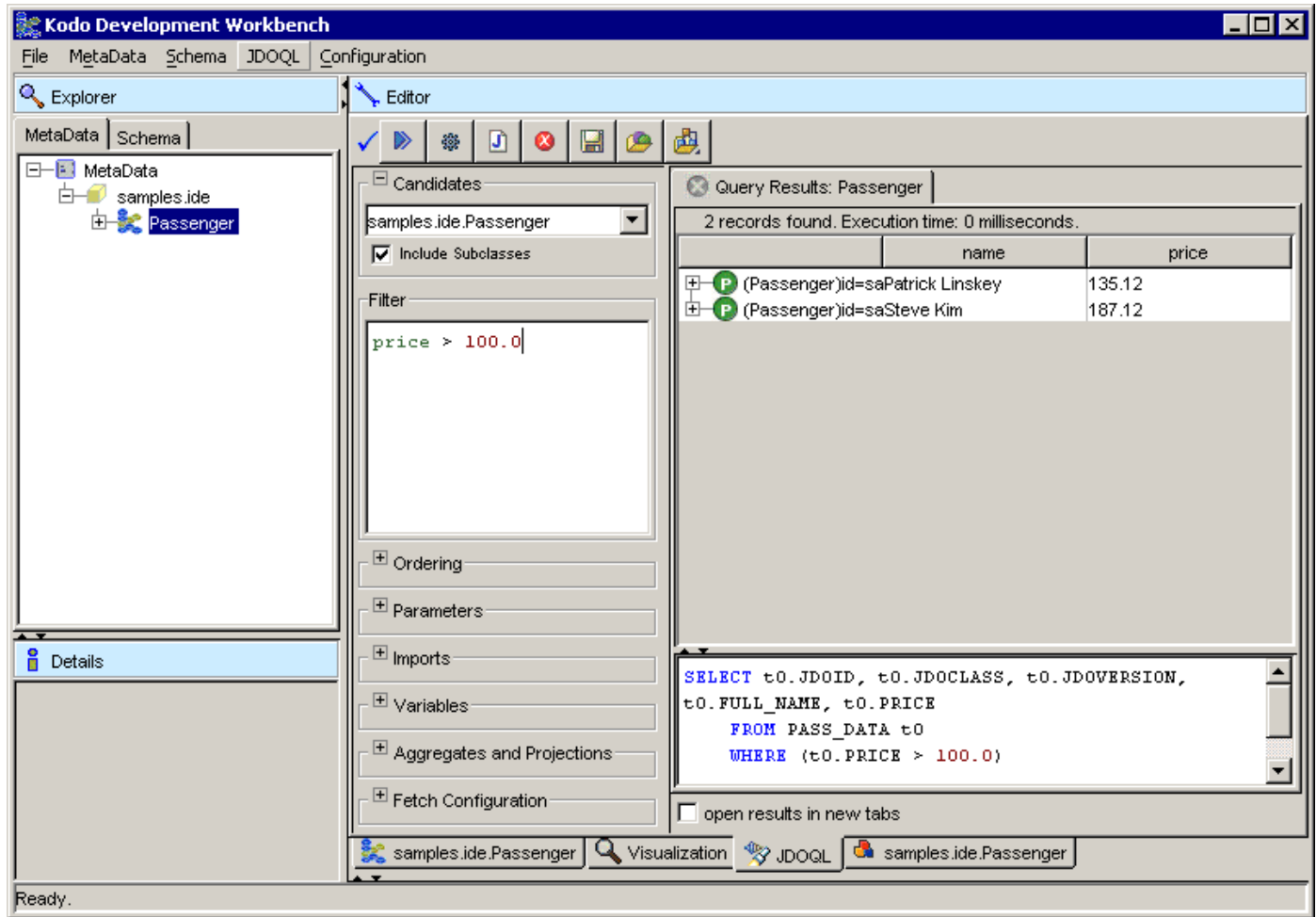
## 2.7.6. JDOQLEditor

The JDOQL Editor in the workbench can be used to experiment with the JDOQL language and browse the results from the database. The editor has full support for all aspects of the JDOQL language, including variables and parameters, as well as aggregates and projections. For a complete description of all the components of the JDOQL Editor, see **Chapter 8, JDOQL Editor [399]**

Now that we have completed mapping the classes to the schema in the tutorial, we can experiment with querying against the existing data in the database. First, select the query editor by selecting "New Query" from the "Query" menu. Next, enter the candidate class, which, in this case, is `samples.ide.Passenger`. Finally, select "Execute Query" from the "Query" menu, and the browser on the right side of the panel should reveal the results from the query, as well as the SQL that was used to execute the query. If you receive any errors, it may be because the mappings were not set up correctly; you should review that all the mappings were defined exactly as specified in the tutorial.



Once you have successfully executed an empty query, you can experiment with executing various different filters. The JDOQL filter editor has syntax highlighting, so that invalid components of the filter (such as a misspelled field) will show up in red. Furthermore, it is possible to "complete" partial field names by entering the first part of a field and hitting the TAB key. If there are any ambiguous matches, then a popup menu will appear allowing you to select which candidate should be used.



## 2.8. Running The Tutorial

Now that our mappings are now correct, we can now exit from Kodo Workbench (File -> Exit), and then run the test program from the prompt:

```
.../samples/ide> java samples.ide.PassengerMain
```

This completes the tutorial. We've seen how to import, edit, and manipulate Kodo objects. The following chapters will include details and examples about the other actions in Kodo Workbench.

---

# Chapter 3. Root MetaData Actions

The Root MetaData Actions represent actions that are system wide.

## 3.1. Mount JDO File

---

This action notifies Kodo Workbench to maintain a .jdo file and add it to the persistent list of metadata files to read on initialization. This will allow the metadata to be edited and managed by Kodo Workbench. However, note that at runtime, the normal rules for **metadata discovery** apply.

## 3.2. Unmount Files...

---

This allows you to unmount .jdo files from the system. This will not delete the file but simply cause the file to be unmanaged. Again, the removed file may still be used by Kodo at runtime.

## 3.3. Create MetaData

---

This action will activate a wizard to create and edit new metadata. The wizard will ask you for all the information needed to write the metadata. Upon completion, the wizard will mount the newly created file and will be treated like every other .jdo file.

The first page will ask what kind of .jdo file you want to create. Note that you can have any combination of the three types in your system just as at runtime. The **JDO Overview's Metadata Placement** chapter describes in detail what each .jdo file can support.

The second panel provides a table listing the classes for which metadata will be created. You can add and remove from the list. Enter a class name into the text field. Valid classes will be in green while invalid entries will be in red. When you are done selecting classes, select Next

The third panel gives you access to the **MetaData Editors** for the new classes. The top drop-down switches the editing panel with the different classes. See the MetaData Editor section for more details on the editor.

When you press finish, Kodo Workbench will proceed to generate and mount the new .jdo files. You can now use the newly create metadata to work on the new classes throughout Kodo Workbench.

## 3.4. Import Mapping Info

---

This allows you to import .mapping files generated by Kodo Workbench or by the **Mapping Tool**. For already mapped classes, this will overwrite the information already stored in the current MappingFactory. This can be done for unmounted classes, although this will *not* cause the metadata to be mounted or created.

---

# Chapter 4. MetaData Actions

## 4.1. Enhance

---

This will cause Kodo's **Enhancer** to process the selected metadata. If classes are already enhanced, nothing further will be done to the corresponding `.class` file.

## 4.2. Edit MetaData

---

This will open the corresponding metadata for the selected classes in the Editor pane. See the **MetaData Editor** section for more details on how the editor works.

## 4.3. Add - Recreate Mapping Info

---

This action will cause the **Mapping Tool** to generate the default mapping info for the selected classes. If mapping info already exists for the class, that information is overwritten with the new info.

The SchemaTool action for the given action will cause tables and columns to be added to Kodo Workbench's virtual schemas. However unused components are left to prevent accidental deletion of schema components used elsewhere.

## 4.4. Export Mapping Info

---

This will export the mapping information for the selected classes to a `.mapping` file. This allows you to store your mapping information as a file resource for source control, runtime use, or source examination. Furthermore, this file can be later used to be imported either the Mapping Tool or through the **Import Mapping Info** action.

## 4.5. Drop Mapping Info

---

This will drop the mapping information for the selected classes from the current MappingFactory. For example in a database-based MappingFactory, this usually means deletion of the corresponding rows. This action corresponds to the `drop` action in the **Mapping Tool**.

## 4.6. Remove MetaData

---

This will remove the selected classes metadata from both Kodo Workbench as well as the corresponding `.jdo` file. If you are attempting to remove the last metadata associated with a given file, Kodo Workbench will give you the additional option of unmounting the file instead of deleting the file altogether. If the option to only unmount the file is selected, the action is comparable to simply using the **Unmount Files...** Root MetaData Action.

## 4.7. Build Schema From Mapping

---

While most applications will usually use Kodo's default mappings or map classes to an existing schema, this option allows Kodo to attempt to build a schema that best corresponds to the mapping information given to it. The **Visualize Mapping** action will give you an idea of what this action will generate. See **Mapping Tool** documentation for more details on what this action will do.

## 4.8. Visualize Mapping

---

Kodo Workbench will generate a visual representation of the mapping information that Kodo will use to persist data at runtime.

Each class and table for the selected mappings will be laid out as nodes of a graph on the canvas. Each class is represented in blue with the mapping icon beside the class name. Each table is in green with the table icon beside the table name. Tables found also in the Workbench schema will also be annotated with its columns' types.

You can edit the mapping information for each selection. By selecting a class, for instance, you can choose a different table to store the data columns. When you select a class or field, you will see detailed information presented in the Details pane. The Details pane will show all the current mapping information for the current selected item. In addition, you will see a selection of colored buttons which you can use to map the current selection. The currently selected option will be highlighted with a yellow border.

You can now select in the detail pane the colored buttons to link the mapping to the proper schema object. For example, by selecting a class, one may see buttons for table and primary key column. By selecting table, one can see that there is now a dashed temporary line leading from the class to the cursor. You can now move the line and "drop" it to select a new table for the class. Similarly, one can similarly select a column name for column attributes. The final type of link attribute are most often found in foreign key mappings. In these links, Kodo Workbench will prompt you for a source column which is being linked to the selected target column. You can see the results in the attribute table and also in the corresponding editor.

You can use foreign keys in a similar fashion to the way you set column and table values for your mappings. In this situation, when dealing with `ref-column` attributes, the visualization editor will populate the table and reference column attributes. When selecting a foreign key for data columns, it will populate the column information that is appropriate for the type of attribute (i.e. data-column, element-column, etc).

You can tailor the amount of information shown on the graph. By right clicking anywhere on the graph and selecting **Edit Detail**, a dialog offering different visualization toggles will appear. Select what portions of the graph you want the system to generate. You can also view a legend of line colors and their meanings by right clicking on the graph and selecting **View Legend**. Each node is draggable by the mouse (click and drag) so that you can re-arrange the layout as you see fit.








Finally, using the right click menu, you can import the Workbench schema to use in your mapping. Select **Add/import Tables**, and then choose to import the entire schema or a subset of schemas/tables. This allows you a quick way to map your classes to your existing schema.




By using the keyboard and putting the scrolling canvas in focus, you can scale and zoom the graph by using the `+/=` key to zoom in and the `-/_` key to zoom out. Press the space bar to return the canvas to normal zoom.

In addition to the table and class nodes, you may also see yellow mapping nodes. For foreign key-based field mappings, Kodo Workbench will generate this node indicating the type of mapping used for the linked field. These nodes are linked back to the owning field by a green line.

There are a number of other lines in the system which each represent a different part of the Kodo mapping system.

**Table 4.1. Graph Edges**

Line Color	Definition
 - Dark Blue	Class Inheritance
 - Dark Green	Class Mapping
 - Black	Datastore Primary Key Column
 - Light Green	Field Mapping
 - Light Blue	Data Column
 - Purple	Key Column
 - Red	Reference Target

Line Color	Definition
 - Magenta	Persistent related field type
 - Yellow	Element persistent type
 - Dark Purple	Key persistent type

You can close the visualization component by closing it like any other editor (File -> Close). The visualization component will also ask you if you want the graph regenerated after one of the visualized class mappings have changed.

---

# Chapter 5. Root Schema Actions

These actions represent actions that operate on the database-wide level, such as synchronizing the actual database with the virtual one in Kodo Workbench, and importing / exporting schema information from file.

## 5.1. Run SchemaTool

---

This will provide a front-end to Kodo's **Schema Tool**. This tool will allow you to update your actual database with the one stored in Kodo Workbench.

The dialog that appears when the action is selected has four major elements. The first two select which action you want to take. While these correspond to those of Schema Tool, they are listed to illustrate what this means in Kodo Workbench:

- **Add** - This will cause Kodo Workbench to only add to the existing database schema. Thus objects that are in Kodo Workbench's schema will be added to the database. However objects in the database but not in Kodo Workbench will be left alone.
- **Refresh** - The Schema Tool will add objects that are in Kodo Workbench's schema that are absent in the database. In addition, components that appear in the database that are not in Kodo Workbench's will be removed. This will cause Kodo Workbench and the actual database schema to become totally synchronized.
- **Create** - The Schema Tool will execute the SQL to recreate Kodo Workbench's schema in the database, regardless of the current existing schema. This is best used when initializing a database.
- **Retain** - The Schema Tool attempt to drop all components in the database that are not present in Kodo Workbench's schema. This will effectively trim the actual database of unnecessary columns and tables.
- **Drop** - The Schema Tool will drop all components that are currently in Kodo Workbench's schema from the database.

The last two options control the details of the Schema Tool's actions which corresponding to the matching command line option. `Ignore Errors?`, when checked, will cause the SchemaTool to ignore any errors that occur in the schema manipulation process. `Drop Unused Tables?` checkbox will tell the SchemaTool to drop tables that appear to be unused.

## 5.2. Refresh Schema From DB

---

This will cause Kodo Workbench to build its schema from that stored in the database. This will cause Kodo Workbench to destroy what was previously stored in the virtual schema. This action is useful when you are starting a new project based on a legacy schema to avoid manually entering and editing the schema information yourself.

## 5.3. Create DB Script

---

This action will generate a file that can be used to generate a brand new schema through the database's own tools such as iSQL or SQL Worksheet. The action will prompt for a file to write to, and upon Ok being selected, will proceed to write the SQL necessary to create Kodo Workbench's schema upon a database.

## 5.4. Create Change Script

---

This action will generate a file that can be used to synchronize your existing schema with Kodo Workbench's schema with your database's own SQL tools. This is useful for tailoring the SQL that Kodo generates by default for special indexes or for DBA verification.

---

## 5.5. Reverse Map Schema

---

This action will present a light front end to the **Reverse Mapping Tool**. Upon selection, a dialog with all of the major options of the tool will be created. Each label is tooltipped with an explanation of what it provides.

To start the tool, first select the file type, package name, and the code directory. Then select any mapping options. Then select Run. Kodo will then run the Reverse Mapping Tool and generate source code, mapping information, and metadata to the specified directory.

### Note

The generated source and metadata will not be available to Kodo Workbench until the classes compiled, Kodo Workbench restarted, and the metadata mounted.

## 5.6. Import .schema file

---

This action will import a `.schema` into Kodo Workbench's current virtual database. These files are either generated from the **Export Root Schema Action**, by hand writing, or by the **Schema Tool**. The existing Kodo Workbench schema will be preserved to the best of its ability dependent on the new data in the file.

## 5.7. Export to .schema file

---

This action will export Kodo Workbench's virtual schema to a file. This file can then be re-imported later into Kodo Workbench, or used in the command line by the **Schema Tool**. This provides a way to separate the schema from both the actual schema and Kodo Workbench for source control, DBA analysis, or for archival uses.

---

# Chapter 6. Schema Actions

These actions represent actions that are based upon the currently selected nodes in the Schema Explorer.

## 6.1. Drop Schema Object

---

This will drop the selected object from Kodo Workbench's virtual schema and leave the actual database untouched.

## 6.2. Edit Table

---

This will open the selected tables for editing in the editing pane. See the **Table Editor** section for further information on how the Table Editor works.

## 6.3. Add

---

This action is available at the Schemas and Schema level. Upon selecting this action, Kodo Workbench will prompt for a new object name. What object is created is dependent on the current selected node. On a Schemas node, a new Schema is generated. And for a Schema object, Kodo Workbench will create a new Table under the selected Schema.

---

# Chapter 7. The Editors

The editors all function on the same basic principals. First, they are all controlled by the `File` main menu, in terms of saving and closing. Second, they all share the same editing pane and can be active and inactive at your control and are identified by an icon, a context name, and potentially a modified flag. The final shared feature of the editors is that they will not store the changes until a save occurs. For example, a column that is removed from a table editor will not be removed from Kodo Workbench's schema until saved. This prevents potentially inaccuracies as tasks are run against changing data model.

## 7.1. MetaData Editor

---

The MetaData editor provides access to all the options specified in the **JDO metadata** specification. Each option has a label which usually has tooltips describing in detail what the option controls and a corresponding input which will set the appropriate value in the metadata.

At the field level, these options are dependent on the type of the field of the class. For example, with `Collection` fields, the editor expands to include a place to enter element-type class name as well as an element-embedded flag option.

In addition, at both levels of the editor is the option to add extensions that are known to Kodo. By selecting an extension from the drop-down, the description box will give further details on what each extension does. Activate each extension by selecting it from the drop-down, pressing add and then entering a value for the newly added row in the `Value` column. You can also add custom vendor/key/value combinations by selecting the `Add` button on the right.

This editing component is also used by the metadata creation wizard, but functions in the same manner as in the editor mode.

## 7.2. The Mapping Editor

---

The Mapping Editor gives the developer access to all of the mapping information stored for the given class. The editor is divided into two parts. The top half corresponds to class-level mapping information. The bottom half corresponds to field mappings for each field in the class.

The top half is further divided by tabs into three mapping components. These three tabs control the currently selected mappings for class data, version indicator and class indicator.

However, both the class-level and field-level components are all based on the same mapping component system. Each component is built of three major parts. The first provides a selector for Kodo-known mappings for the given type, be it a field, class, or version mapping. When you select a mapping type from the drop down, the second component, the mapping description, will update with a thorough explanation of what the current selected mapping type does. And the final component provides inputs for each mapping type.

The inputs for a given mapping type are usually of three major types:

- *Single schema selection* - These provide a simple textfield to enter a schema object of the appropriate type. They are accompanied by a button to select graphically from Kodo Workbench's schema representation in a component not unlike the Schema explorer.
- *Optional schema selection* - These function similar to the single schema selection inputs, however they are controlled further by a checkbox which enables or disables the input value. For example, `Collections` are not ordered by default. By selecting the `Ordered` checkbox, and entering a column name to store the order, the `Collection` field mapping will know to store the order in the specified column at runtime.
- *Foreign Key selection* - These inputs allow the entering of the three major elements in a foreign key: table, source columns, and target columns. Each is dependent on the kind of foreign key being built, however they provide 2 major components, an single schema selection input for the table, and a table to store source and target column pairs. These column pairs can be added and removed by the controls on the right. Furthermore, each cell's value can be entered by a graphical selection from Kodo Workbench's schema by using the `From Database` button.

A mapping type available at every level is the `Custom` mapping type. This requires a high level of understanding of the Kodo **mapping system** and provides a streamlined access to the information stored in the `MappingInfo` for the class.

## 7.3. The Visualization Editor

---

This editor as noted in the **Visualization** action documentation. You can save changed mapping information by saving it like the other editors.

## 7.4. The Table Editor

---

This is a fairly straightforward table editor. This editor exposes graphical ways of adding / removing columns, changing primary key columns, and schema details of each table. These changes occur at Workbench schema level and won't take effect until the next synchronization via the **Schema Tool**

## 7.5. The Foreign Keys Editor

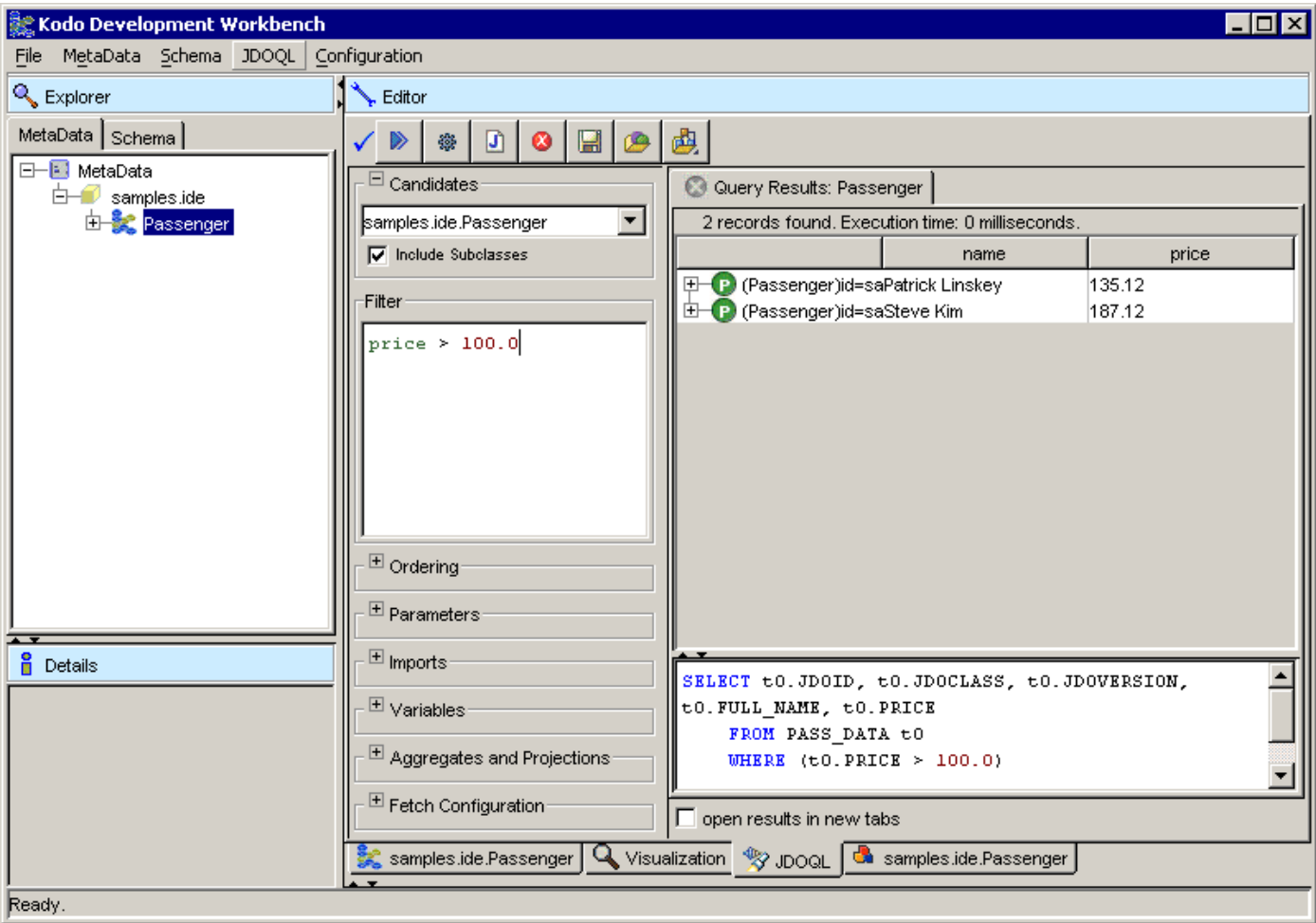
---

This editor provides a way to view, change, and create foreign keys in the Workbench's schema. Once created, you can then later add them to your real database by using the **Schema Tool**.

# Chapter 8. JDOQL Editor

The JDOQL Editor provides a means to compose, validate, and test queries against your database. This is useful both in terms of validating your metadata and mappings for your classes, as well as providing a simple graphical mechanism for browsing the contents of your database and traversing relations.

The JDOQL Editor consists of two components: on the left is set of components that can be used to build up the query. On the right side is information about the results of the query: the SQL that was executed in order to execute the query, as well as tree that provides information on the fields that were retrieved from the database.



This section describes the various GUI components that are used to build up queries. For more details about the meaning of the various elements of JDO queries, see [Chapter 11, Query \[54\]](#).

## 8.1. Query Validator

The toolbar in the upper-left corner of the JDOQL editor contain a series of buttons that are used to build up and execute the query. The leftmost component an icon that will be either a yellow warning icon, which indicates that the the query is invalid, or a checkbox, which indicates that the query is syntactically valid. When the query is invalid for any reason, placing the mouse over the warning icon will provide some information in the tooltip text about why the query is not valid, which can assist in fixing the query. Note that even when the query is syntactically valid, executing the query may still cause an error (in which case a window will open describing the error). For example, this might happen if the mappings for the class are set up incorrectly, such as when a class is mapped to a non-existent table.

## 8.2. Candidate Class Editor

---

The Candidate Class Editor is where the candidate class for the filter can be entered. The text in the field will be red if the class is not valid. Furthermore, the drop-down will be populated by all the classes that the system current knows about.

Beneath the class editor is a checkbox marked "include subclasses", which specifies whether or not to include subclasses in the query.

## 8.3. Filter Editor

---

The filter editor is the primary component for building up queries. The editor performs syntax highlighting, by identifying valid and invalid components of a JDO query. Furthermore, the query editor allows for the automatic completion of fields, methods, variables, and parameters using the "tab" key. For example, when querying against a class that has a field called "firstName", you can type the letter "f" and then hit the "tab" key, which will cause the editor to fill in "firstName". If there are an ambiguous completions, a menu will pop up to allow you to select which completion to use. This provides a convenient way to enter fields and traverse relations without having to memorize the exact field names in each of the classes involved.

## 8.4. Additional Query Component Editors

---

The remaining components of the query all reside in collapsable borders that can be expanded or collapsed by clicking on the arrow in the title of the component. For example, if you click on the arrow next to the "Imports" label, the imports editor will expand to allow editing.

The editors for each of these components have a plus and minus icon, for use in adding or removing elements from the component. Furthermore, double-clicking in any blank space in the component will add a new row that can then be edited.

### 8.4.1. Ordering Editor

---

The ordering editor provides a means of specifying the ordering of the results of the query. The "field" column will contain the name of the field on which to order, and the "direction" field will be either "ascending" or "descending".

### 8.4.2. Parameters Editor

---

The parameters editor allows the user to specify one or more parameters for the query to use. The "type" of the parameter is the class of the parameter, the "name" of the parameter is the name by which the parameter will be referred in the filter editor, and the value is the value of the field. For example, if you add a new row with the type set as "String", and name set as "fnameParam", and the value set to "Marc", then in the filter editor you could enter the filter (against some class that has a field named "firstName") `firstName == fnameParam`. The query will then be executed as if the filter had read `firstName == "Marc"`. For more details on using parameters in queries, see [Chapter 11, Query \[54\]](#).

Some other features of the parameter editor are:

- Strings and numbers are entered literally.
- Collection parameters can be specified as being separated by commas. For example, your filter could read `namesParam.contains(firstName)`, and the parameter could be entered with the type set to "java.util.List", the name set to "namesParam", and the value set to "Steve,Abe". This query will find all people whose first name is "Steve" or "Abe".
- Persistent parameters (i.e., parameters that are of a persistent type) are entered using the stringified object id of the type. For example, if the type of the parameter is "Person", the name of the parameter is "personParam", and the value of field is "com.mycompanyt.Person-14760", then the filter `employees.contains(personParam)` (executed against some class that has a collection of Person instances called "employees") will execute the query with that persistent parameter.
- It is possible to drag a row from the JDO Results browser into the parameters table, which will cause a new parameter to be

declared with that object id. For example, if the results browser is showing a Person instance, dragging that instance will declare a new parameter with type set to "Person", name set to an auto-assigned name "personParam", and the value set to the stringified object id of the instance. This can greatly ease the tedium of declaring persistent parameters, since you do not need to type in what can be a lengthy stringified object id.

### 8.4.3. Imports Editor

---

The imports editor allows you to declare imports so that you do not need to enter the complete package name in the variables or parameters editor. For example, if you enter a row with `com.mycompany.*`, you can then just enter `Company` as the type of a parameter without having to fully qualify the name. Another handy use is to add `java.util.*`, so that you do not need to fully qualify collection parameter declarations (such as `java.util.Set`).

### 8.4.4. Variables Editor

---

The variables editor allows you to declare variables to use in the query. For example, when executing against some "Company" class that has a collection called "employees" that contains instances of "Employee", you can declare a variable of type "Employee" and name "emp", and the filter can be entered as `employees.contains(emp) && emp.firstname == "Patrick"`, which will find all Company instances that have an employee whose first name is "Patrick".

### 8.4.5. Aggregates and Projections Editor

---

The aggregates and projections editor allow you to edit these features of the query. For details on using aggregates and projections in queries, see [Chapter 11, Query \[54\]](#). For example, you can obtain only the "name" fields by executing a query against a Company with the "field/expression" row of the Aggregates and Projections set to "name" (the "alias" field can optionally be left blank). Another example is that you can execute a query against a "Person" class with an aggregate row set to `sum(salary)`.

### 8.4.6. Fetch Configuration Editor

---

The fetch configuration editor allows you to specify the fetch groups that will be used when executing the query. For more details about fetch configuration, see [Section 10.5, "Fetch Configuration" \[290\]](#).

## 8.5. Execute Query

---

The "Execute Query" menu item and button will execute the currently defined query and show the results (as well as the SQL used to execute the query) in the results browser (see [Section 8.12, "Results Browser" \[402\]](#)). This option will be grayed out if the current query is syntactically invalid. Note that this option has a shortcut of Control-Enter, which is useful to rapidly re-execute a query after making minor changes to the filter.

## 8.6. Show SQL

---

The "Show SQL" menu item/button will show only the SQL that will be used to execute the query. This is useful to fine tune query parameters and fetch configuration for your queries.

## 8.7. Clear Query

---

The "Clear Query" menu item/button will clear out all the fields of the query editor, allowing you to start with a clean slate.

## 8.8. Show Java

---

The "Show Java" menu item/button will show the java code that can be used to build up the complete query as defined in the edit-

or. This is useful for copying & pasting into your application code once you have your query defined.

## 8.9. Save Query

---

The "Save Query" menu item/button saves the current contents of the query into a file that you will specify. This can later be loaded by the "Load Query" item. Note that this is not java or SQL code, but a binary representation of the contents of the current query filter.

## 8.10. Load Query

---

This "Load Query" menu item/button allows you to load the previously saves contents of the query editor.

## 8.11. Recent Queries

---

The "Recent Queries" sub-menu contains recently executed queries, so that you can go back and re-execute queries that you have already seen.

## 8.12. Results Browser

---

The results browser will be displayed whenever a query is executed with the "Execute Query" option (see [Section 8.5, “Execute Query” \[401\]](#)). The results is a combination of a tree and a table. The table contains a read-only view of all the primitive or String fields of the candidate instances that result from the execution of the query. On the leftmost side is a tree pivot that allows you to "dig" into the object, revealing individual rows for each of the fields of the object. Child collection fields can also be opened, which allows you to traverse your persistent object graph to an arbitrary depth. The browser is useful to see the exact results from the database, as well as ensuring that your mappings for relations are set up correctly.

It is possible to drag persistent entries from the class into various components of the JDOQL editor. For details, see [Section 8.4.2, “Parameters Editor” \[400\]](#).

---

# Appendix A. 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 B. Supported Databases

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

*Table B.1. Supported Databases and JDBC Drivers*

Database Name	Database Version	JDBC Driver Name	JDBC Driver Version
Apache Derby	10.0.2.1	Apache Derby Embedded JDBC Driver	10.0.2.1
Borland Interbase	7.1.0.202	Interclient	4.5.1
Borland JDataStore	6.0	Borland JDataStore	6.0
DB2	8.1	IBM DB2 JDBC Universal Driver	1.0.581
Empress	8.62	Empress Category 2 JDBC Driver	8.62
Firebird	1.5	JayBird JCA/JDBC driver	1.0.1
Hypersonic Database Engine	1.7.0	Hypersonic	1.7.0
Informix Dynamic Server	9.30.UC10	Informix JDBC driver	2.21.JC2
InterSystems Cache	5.0	Cache JDBC Driver	5.0
Microsoft Access	9.0 (a.k.a. "2000")	DataDirect SequeLink	5.4.0038
Microsoft SQL Server	8.00.194 (SQL Server 2000)	SQLServer	2.2 (2.2.0002)
Microsoft Visual FoxPro	7.0	DataDirect SequeLink	5.4.0038
MySQL	3.23.43-log	MySQL Driver	3.0.14
Oracle	8.1-9.2	Oracle JDBC driver	9.0 (9.0.1.0.0)
Pointbase	4.4	Pointbase JDBC driver	4.4 (4.4)
PostgreSQL	7.2.1	PostgreSQL Native Driver	7.2 (7.2)
Sybase Adaptive Server Enterprise	12.5	jConnect	5.5 (5.5)

## B.1. Apache Derby

---

*Example B.1. Example properties for Derby*

```
javax.jdo.option.ConnectionDriverName: org.apache.derby.jdbc.EmbeddedDriver
javax.jdo.option.ConnectionURL: jdbc:derby:DB_NAME;create=true
```

### B.1.1. Known issues with Derbyt

---

- Apache Derby is still in pre-release (or "incubator" stage), so future versions may introduce incompatibilities with the current

dictionary.

## B.2. Borland Interbase

---

### *Example B.2. Example properties for Interbase*

```
javax.jdo.option.ConnectionDriverName: \
interbase.interclient.Driver
javax.jdo.option.ConnectionURL: \
jdbc:interbase://SERVER_NAME:SERVER_PORT/DB_PATH
```

### B.2.1. Known issues with Interbase

---

- Interbase does not support record locking, so datastore transactions can use the pessimistic lock manager.
- Interbase does not support the LOWER, SUBSTRING, or INSTR SQL functions, which means that `toLowerCase()`, `indexOf()`, and `substring()` methods in JDOQL cannot be used.

## B.3. JDataStore

---

### *Example B.3. Example properties for JDataStore*

```
javax.jdo.option.ConnectionDriverName: \
com.borland.datastore.jdbc.DataStoreDriver
javax.jdo.option.ConnectionURL: \
jdbc:borland:dslocal:db-jdatastore.jds;create=true
```

## B.4. IBM DB2

---

### *Example B.4. 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
```

---

## B.4.1. Known issues with DB2

---

- Floats and doubles may lose precision when stored.
- Empty char values are stored as NULL.
- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending `DB2Dictionary`.

## B.5. Empress

---

### *Example B.5. Example properties for Empress*

```
javax.jdo.option.ConnectionDriverName: empress.jdbc.empressDriver
javax.jdo.option.ConnectionURL: jdbc:empress://SERVER=yourserver;PORT=6322;DATABASE=yourdb
```

## B.5.1. Known issues with Empress

---

- Empress enforces pessimistic semantics (lock on read) when not using `AllowConcurrentRead` property (which bypasses row locking) for `EmpressDictionary`.
- Only the category 2 non-local driver is supported.

## B.6. Hypersonic

---

### *Example B.6. Example properties for Hypersonic*

```
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionURL: jdbc:hsqldb:DB_NAME
```

## B.6.1. Known issues with Hypersonic

---

- Hypersonic does not properly support foreign key constraints.
- Hypersonic does not support pessimistic locking, so non-optimistic transactions will fail unless the `SimulateLocking` property is set for the `kodo.jdbc.DBDictionary`

## B.7. Firebird

---

### *Example B.7. Example properties for Firebird*

```
javax.jdo.option.ConnectionDriverName: org.firebirdsql.jdbc.FBDriver
javax.jdo.option.ConnectionURL: jdbc:firebirdsql://SERVER_NAME:SERVER_PORT/DB_PATH
```

### B.7.1. Known issues with Firebird

---

- The Firebird JDBC driver does not have proper support for batch updates, so batch updates are disabled.
- Firebird does not support auto-increment columns.
- Firebird does not support the LOWER, SUBSTRING, or INSTR SQL functions, which means that `toLowerCase()`, `indexOf()`, and `substring()` methods in JDOQL cannot be used.

## B.8. Informix

---

### *Example B.8. 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
```

### B.8.1. Known issues with Informix

---

•

## B.9. InterSystems Cache

---

### *Example B.9. Example properties for InterSystems Cache*

```
javax.jdo.option.ConnectionDriverName: com.intersys.jdbc.CacheDriver
javax.jdo.option.ConnectionURL: jdbc:Cache://SERVER_NAME:SERVER_PORT/DB_NAME
```

## B.9.1. Known issues with InterSystems Cache

---

- Support for Cache is done via SQL access over JDBC, not through their object database APIs.

## B.10. Microsoft Access

---

*Example B.10. Example properties for Microsoft Access*

```
javax.jdo.option.ConnectionDriverName: com.ddtek.jdbc.sequelink.SequeLinkDriver
javax.jdo.option.ConnectionURL: jdbc:sequelink://SERVER_NAME:SERVER_PORT
```

### B.10.1. Known issues with Microsoft Access

---

- Using the Sun JDBC-ODBC bridge to connect is not supported.

## B.11. Microsoft SQL Server

---

*Example B.11. Example properties for Microsoft 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;sendStringParametersAsUnicode=false
```

### B.11.1. Known issues with SQL Server

---

- SQL Server date fields are accurate only to the nearest 3 milliseconds, possibly resulting in precision loss in stored dates.
- The ConnectionURL must always contain the "selectMethod=cursor" string.
- Adding `sendStringParametersAsUnicode=false` to the ConnectionURL may significantly increase performance.
- The Microsoft SQL Server driver only emulates batch updates. The DataDirect JDBC driver has true support for batch updates, and may result in a significant performance gain.
- Floats and doubles may lose precision when stored.

- TEXT columns cannot be used in queries.

## B.12. Microsoft FoxPro

---

### *Example B.12. Example properties for Microsoft FoxPro*

```
javax.jdo.option.ConnectionDriverName: com.ddtek.jdbc.sequellink.SequeLinkDriver
javax.jdo.option.ConnectionURL: jdbc:sequelink://SERVER_NAME:SERVER_PORT
```

### B.12.1. Known issues with Microsoft FoxPro

---

- Using the Sun JDBC-ODBC bridge to connect is not supported.

## B.13. MySQL

---

### *Example B.13. Example properties for MySQL*

```
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL: jdbc:mysql://SERVER_NAME/DB_NAME
```

### B.13.1. Known issues with MySQL

---

- The default table types that MySQL uses do not support transactions, which will prevent Kodo from being able to roll back transactions. The table type of "InnoDB" should be used for any tables that Kodo will be used.
- MySQL does not support sub-selects in versions prior to 4.1, and are disabled by default. Some operations (such as the `isEmpty()` method in a JDOQL query) will fail due to this. If you are using MySQL 4.1 or later, you can lift this restriction by setting the `SupportsSubselect=true` parameter of the **kodo.jdbc.DBDictionary** property.
- 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.
- Some version of the MySQL JDBC driver have a bug that prevents Kodo from being able to interrogate the database for for-

eign keys. Version 3.0.14 (or higher) of the MySQL driver is required in order to get around this bug.

## B.14. Oracle

---

### *Example B.14. Example properties for Oracle*

```
javax.jdo.option.ConnectionDriverName: oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionURL: jdbc:oracle:thin:@SERVER_NAME:1521:DB_NAME
```

### B.14.1. Known issues with Oracle

---

- For VARCHAR fields, null and a blank String are equivalent. This means that an object that stores a null String field will have it get read back as a blank String.
- Oracle corp's JDBC driver for Oracle has only limited support for batch updates. The result for Kodo is that in some cases, the exact object that failed an optimistic lock check cannot be determined, and Kodo will throw an `JDOOptimisticVerificationException` with more failed objects than actually failed.
- Oracle cannot store numbers with more than 38 digits in numeric columns.
- Pessimistic locking is not supported on queries that use `SELECT DISTINCT`. 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.

## B.15. Pointbase

---

### *Example B.15. 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=30720
```

### B.15.1. Known issues with Pointbase

---

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

## B.16. PostgreSQL

---

### *Example B.16. Example properties for PostgreSQL*

```
javax.jdo.option.ConnectionDriverName: org.postgresql.Driver
javax.jdo.option.ConnectionURL: jdbc:postgresql://SERVER_NAME:5432/DB_NAME
```

### B.16.1. Known issues with PostgreSQL

---

- Pessimistic locking is not supported on queries that use SELECT DISTINCT. 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.

## B.17. Sybase Adaptive Server

---

### *Example B.17. Example properties for Sybase*

```
javax.jdo.option.ConnectionDriverName: com.sybase.jdbc2.jdbc.SybDriver
javax.jdo.option.ConnectionURL: \
    jdbc:sybase:Tds:SERVER_NAME:4100/DB_NAME?ServiceName=DB_NAME&BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true
```

### B.17.1. Known issues with Sybase

---

- The "DYNAMIC\_PREPARE" parameter of the Sybase JDBC driver cannot be used with Kodo.
- 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.
- Persisting a zero-length string results in a string with a single space characted being returned from Sybase, Inc.'s JDBC driver.
- The BE\_AS\_JDBC\_COMPLIANT\_AS\_POSSIBLE is required in order to use datastore (pessimistic) locking. Failure to set this property may lead to obscure errors like "FOR UPDATE can not be used in a SELECT which is not part of the declaration of a cursor or which is not inside a stored procedure."



---

# Appendix C. Common Database Errors

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

*Table C.1. Known Database Error Codes*

Database	Error Code	SQL State	Message	Solution
DB2	-803	23505	<b>SQL0803N</b> 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	<b>SQL0511N</b> 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	<b>SQL0401N</b> 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	<b>SQL0302N</b> 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 mapping-tool, ensure that the jdbc-size JDO metadata extension specifies a large enough size for the column.
DB2	-204	42S02	<b>SQL0204N</b> "%s" is an undefined	The database schema does not match the map-

Database	Error Code	SQL State	Message	Solution
			name .	ping defined in the metadata for the persistent class. See the <b>mapping</b> documentation.
DB2	-99999	22003	Numeric value out of range.	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to hold 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. Make sure to enable <b>SQL statement ordering</b> .
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: S1000 General error java.lang.NumberFormatException: %d in statement %s	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. 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	<b>1196</b>	S1000	General error:	One or more tables that

Database	Error Code	SQL State	Message	Solution
			Warning: Some non-transactional changed tables couldn't be rolled back	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 <code>TableType</code> property of the <b>DBDictionary</b> configuration 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 <b>MySQL deadlock</b> 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 data source, connection testing can be configured via the <b>ConnectionFactoryProperties</b> 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 <code>Double.POSITIVE_INFINITY</code> or <code>Double.NEGATIVE_INFINITY</code> values.
Oracle	17069	null	Use explicit XA	Manual transaction op-

Database	Error Code	SQL State	Message	Solution
			call	erations were attempted on a data source that was configured to use an XA transaction. In order to utilize XA transactions, set the <b>kodo.jdbc.DataSourceMode</b> property to en-listed.
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 <code>kodo.properties</code> , nor was it specified in the database configuration mechanism, nor was it specified in the <code>PersistenceManager.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 <b>mapping</b> 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"> <li>1. Increase the number of cursors allowed in the data-</li> </ol>

Database	Error Code	SQL State	Message	Solution
				<p>base. This is typically done by increasing the <code>open_cursors</code> parameter in the <code>initSID-NAME.ora</code> file.</p> <ol style="list-style-type: none"> <li>2. Ensure that Kodo query results and extent iterators are being closed, since open results will maintain an open <code>ResultSet</code> on the server side until they are garbage collected.</li> <li>3. Decrease the value of the <code>Max-CachedStatements</code> parameter in the <b>ConnectionFactory-Properties</b> configuration property.</li> </ol>
Oracle	932	42000	ORA-00932: inconsistent data-types: 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 <code>jdbc-size</code> metadata extension to -1.
Oracle	1	23000	ORA-00001: unique constraint (%s) violated	Duplicate values have been inserted into a column that has a <code>UNIQUE</code> 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 <code>NUMERIC</code>

Database	Error Code	SQL State	Message	Solution
				fields have a limitation of 38 digits.
Oracle	0	null	Overflow 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.
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

Database	Error Code	SQL State	Message	Solution
				column.
SQLServer	0	08007	Can't start a cloned connection while in manual transaction mode.	Append <code>";SelectMethod=cursor"</code> to the ConnectionURL. See the description of the problem on the <a href="#">Microsoft support site</a> .
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 prepared statement. 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 list. Columns in the order by list must be unique.	Ensure that there are no duplicates in the ordering of the query.

Database	Error Code	SQL State	Message	Solution
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 data source, connection testing can be configured via the <b>ConnectionFactoryProperties</b> configuration property.
Sybase	<b>311</b>	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 mappingtool 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	<b>2762</b>		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 <b>sp_dboption database_name,"ddl in tran", true</b> from isql. See the Sybase documentation for <b>allowing data definition commands in transactions</b> .
Sybase	0	JZ0BE	JZ0BE: BatchUpdateException: Error occurred while executing batch statement: Arithmetic overflow during implicit conversion of NUMERIC value '%d' to a NUMERIC field.	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	0	JZ00B	JZ00B: Numeric overflow.	A numeric overflow occurred. Ensure that the capacity of the numeric column is sufficient to

Database	Error Code	SQL State	Message	Solution
				store the specified value the persistent object is attempting to store.
Sybase	257	42000	Implicit conversion from data-type '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.	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 mapping-tool, ensure that the jdbc-size JDO metadata extension specifies a large enough size for the column.
Sybase	<b>2601</b>	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.

---

# Appendix D. Upgrading Kodo

This document describes how to upgrade to a new version of Kodo from the prior version. When skipping versions, please follow the steps outlined as though you are converting to each intermediate version.

## D.1. Migrating from Kodo 2 to Kodo 3

---

This document describes how to migrate from Kodo 2 to Kodo 3. It assumes that you are using Kodo 2.4 or above. Unlike Kodo 2.x point releases, Kodo 3 is a major new release that introduces a lot of new functionality, but also changes many aspects of Kodo development. Kodo 3 absolutely will not work with Kodo 2 data unless you complete the steps below in the order that they appear.

### Warning

We strongly recommend that you back up all Java code, JDO metadata, and Kodo configuration files before proceeding.

### D.1.1. Source Code Migration

---

The first and most difficult task in migrating to Kodo 3 is to compile all of your code against the Kodo 3 codebase. In addition to making Kodo more robust, flexible, and feature-packed, one of the primary goals of Kodo 3 was to clean up Kodo's APIs, making it easier for advanced users to understand and manipulate Kodo internals. Unfortunately, these API changes will break existing code that accesses Kodo-specific classes (as opposed to standard JDO interfaces). We cannot create a tool to automatically migrate your code for you, but we hope that our new **Javadoc** and this manual will make any necessary code changes as easy as possible. Our developer newsgroups and **email support** are also available for migration questions.

#### D.1.1.1. Package Structure Changes

---

Kodo 3's package structure has been reworked to be shorter and more logical. The table below includes some of the more commonly-used Kodo 2 packages and their Kodo 3 equivalents.

*Table D.1. Notable Package Changes*

Kodo 2 Package	Kodo 3 Package
com.solarmetric.kodo.runtime	kodo.runtime
com.solarmetric.kodo.conf	kodo.conf
com.solarmetric.kodo.query	kodo.query
com.solarmetric.kodo.meta	kodo.meta
com.solarmetric.kodo.runtime.datacache	kodo.datacache
com.solarmetric.kodo.runtime.datacache.plugins	kodo.datacache
com.solarmetric.kodo.runtime.datacache.query	kodo.datacache
com.solarmetric.kodo.runtime.event	kodo.event
com.solarmetric.kodo.runtime.event.impl	kodo.event
com.solarmetric.kodo.impl.jdbc	kodo.jdbc
com.solarmetric.kodo.impl.jdbc.ormapping	kodo.jdbc.meta
com.solarmetric.kodo.impl.jdbc.schema.dict	kodo.jdbc.sql

### D.1.1.2. API Changes

---

In addition to package restructuring, many Kodo APIs have changed. The list below describes the differences in some of the more significant Kodo classes and APIs.

- `com.solarmetric.kodo.jdbc.JDBCConfiguration` and `com.solarmetric.kodo.jdbc.JDBCSimpleConfiguration` have been replaced by `kodo.jdbc.conf.JDBCConfiguration` and `kodo.jdbc.conf.JDBCConfigurationImpl`, respectively. The Kodo 3 versions are very similar to the Kodo 2 ones, the only major difference being that Kodo 3 does not require a `Connector` argument to any methods, making it easier to access plugins.
- `com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory` and `com.solarmetric.kodo.impl.jdbc.ee.EEPersistenceManagerFactory` have also been replaced. You may refer to one or both of these factories in your Kodo 2 application. In Kodo 3, you will only use the `kodo.jdbc.runtime.JDBCPersistenceManagerFactory`. Whether or not this factory dispenses managed persistence managers by default is controlled by the `kodo.TransactionMode` configuration property. The factory also has versions of the `getPersistenceManager` method that allow you to override the factory's default setting and retrieve managed or unmanaged persistence managers from the same factory instance as needed.
- The `com.solarmetric.kodo.impl.jdbc.schema.dict.DBDictionary` has moved to `kodo.jdbc.sql.DBDictionary`, and its API has changed significantly. As before, we have included the source code to our dictionaries in your Kodo distribution. Note that Kodo 3 automatically avoids all naming conflicts when creating new tables, columns, and indexes by first reflecting on the existing schema to see what names are already in use. So if you were customizing a Kodo 2 dictionary just to avoid naming conflicts, you should be able to simply use a bundled dictionary in Kodo 3.
- `com.solarmetric.util.ObjectIds.Id` has been replaced by `kodo.util.Id`.
- Kodo's **logging channels** have been renamed to match the new package structure.
- The Reference Guide demonstrates the new ways to **access JDBC connections** and **use the sequence factory** at runtime.
- Kodo 3 has new Ant tasks for all included development tools. These tasks are described in **Section 15.2, “Apache Ant” [346]**. As you change your build scripts, keep in mind that while Kodo 3 has something called the schema tool, it is very different than the Kodo 2 schema tool. For all intents and purposes, the Kodo 3 mapping tool replaces the Kodo 2 schema tool. You will probably never need to use Kodo 3's version of the schema tool directly.
- Dependent objects are now deleted as soon as they are removed from their owning object. For example, if you have a `Map` field that you have annotated with the `value-dependent` metadata extension, calling `Map.remove` will cause the value associated with the removed key to transition to the persistent-deleted state immediately. See **Section 6.2.1.3, “dependent” [198]** for details.

Additionally, in Kodo 3 we have added officially-supported interfaces for our API extensions to many of the core JDO classes. The following interfaces will be of particular interest:

- `kodo.runtime.KodoPersistenceManagerFactory`
- `kodo.runtime.KodoPersistenceManager`
- `kodo.runtime.KodoExtent`
- `kodo.query.KodoQuery`

## D.1.2. JDO Metadata Migration

---

The next step in switching to Kodo 3 is to migrate your JDO metadata. This is necessary for two reasons. First, Kodo 3 changes many metadata extension keys to differentiate between back-end independent extensions and jdbc-specific extensions. Second, and more importantly, Kodo 3 changes the way object/relational mapping data is stored.

In Kodo 2, object/relational mapping data was read from JDO metadata extensions. If no mapping extensions existed, then the system assumed that you wanted to use the "default" mapping, and dynamically created one at runtime. While this process was convenient, it was not very robust, because no validation of mappings against the schema was performed, and there was no record of mappings to consult for conflict avoidance. Furthermore, Kodo could never improve on its mapping defaults because anyone who relied on them would be left with an invalid schema.

Kodo 3 remedies these problems by always recording all mapping data, rather than relying on unspecified defaults. This leads to better validation, fewer conflicts, and more robust handling of Java class changes. As with Kodo 2, you can write mapping data yourself, or rely on the system to do it for you. As **you will see**, we offer several choices of where to record mapping information, from JDO metadata extensions to XML mapping files to storing it in a special database table.

The Kodo 2 migrator tool can automatically migrate all of your JDO metadata. You can invoke the tool via the included `kodo2migrator` script or via its Java class, `kodo.jdbc.migration.kodo2.Kodo2Migrator`. The tool also comes with an Ant task, which we present later in this section.

### *Example D.1. Using the Kodo 2 Migrator*

```
kodo2migrator *.jdo
```

The migrator accepts the standard set of command-line arguments defined by the **configuration framework**, along with the following flags:

- `-file/-f <stdout | output file or resource>` : The path or resource name of a file in which to store the mapping information linking your existing persistent classes to your schema. By default, the migrator creates mapping files that mirror the structure of your JDO metadata files.

Each additional argument to the tool should be the class name, `.class` file, `.java` file, or `.jdo` file of a persistent type. **The type must be compiled.** The existing metadata file for the type will be backed up to `<file-name>~`, and a new JDO metadata file will be generated in its place. Object-relational mapping data will be calculated and written to separate `.mapping` files mirroring the structure of your `.jdo` files, or to the file you specified in the `-file` command-line argument.

You should run the Kodo 2 migrator tool on all of your persistent classes at once. One easy way to do this is through the tool's Ant task, `kodo.jdbc.ant.Kodo2MigratorTask`. The attributes of the task correspond exactly to the long versions of the command-line arguments accepted by the tool. Additionally, see the chapter on **Ant integration** for common configuration options available to all Kodo tasks.

### Note

The Kodo 2 migration task has dependencies on both old and new Kodo code, and therefore is very sensitive to the invocation environment. The easiest way to ensure the correct environment is to include your classes and the directory containing your Kodo 2.x `kodo.properties` file in your system `CLASSPATH`.

### *Example D.2. Invoking the Kodo 2 Migrator from Ant*

```
<target name="migrate-meta">
  <!-- define the kodo2migrator task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="kodo2migrator" classname="kodo.jdbc.ant.Kodo2MigratorTask"/>

  <!-- invoke migrator on all .jdo files below the src directory -->
  <kodo2migrator>
    <fileset dir="${basedir}/src">
      <include name="**/*.jdo" />
    </fileset>
  </kodo2migrator>
</target>
```

## D.1.3. Properties File Migration

---

After migrating your JDO metadata, you can migrate your Kodo configuration properties files. You must do this *after* migrating your JDO metadata so that the migrator tool can draw on your Kodo 2 mapping preferences.

The Kodo 2 properties tool can automatically migrate your configuration files. You can invoke the tool via the included `kodo2properties` script or via its Java class, `kodo.jdbc.kodo2.Kodo2Properties`. The tool also comes with an Ant task, which we present later in this section.

### *Example D.3. Using the Kodo 2 Properties Tool*

```
kodo2properties kodo.properties
```

The tool accepts the standard set of command-line arguments defined by the **configuration framework**, along with the following flags:

- `-file/-f <stdout | output file or resource>` : The path or resource name of the migrated properties file. You should only use this argument if you want to override the default behavior of overwriting the original file (after backing it up), and only if you are invoking the tool on only a single properties file.

Each additional argument to the tool should be a Kodo 2 properties file. Unless a `-file` argument is given, the file will be backed up to `<file-name>~`, and a new properties file will be generated in its place.

The Kodo 2 properties tool can also be invoked via its Ant task, `kodo.jdbc.ant.Kodo2PropertiesTask`. The attributes of the task correspond exactly to the long versions of the command-line argument accepted by the tool. Additionally, see the chapter on **Ant integration** for common configuration options available to all Kodo tasks.

### *Example D.4. Invoking the Kodo 2 Properties Tool from Ant*

```
<target name="migrate-props">
  <!-- define the kodo2properties task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="kodo2properties" classname="kodo.jdbc.ant.Kodo2PropertiesTask"/>

  <!-- invoke tool on all the conf/kodo*.properties files -->
  <kodo2properties>
    <fileset dir="${basedir}/conf">
      <include name="**/kodo*.properties" />
    </fileset>
  </kodo2properties>
</target>
```

```
</fileset>
</kodo2properties>
</target>
```

## D.1.4. Storing Object-Relational Mapping Data

---

Please read the reference guide's description of the available mapping factories in [Section 7.3, “Mapping Factory” \[214\]](#). If you are satisfied with the default of storing mapping data in `.mapping` files, then there is nothing to do. If you'd like to switch to another mapping data format, however, the above chapter describes your options and shows you how to migrate your mapping data between factories.

You are now ready to develop with Kodo 3.

## D.1.5. Kodo 3 Development Process

---

The development process under Kodo 3 is slightly different than it was under Kodo 2. For example, the familiar `schematool` has been replaced with the more powerful `mappingtool`. You should take a moment to read through the revised [Kodo tutorials](#) and the [Reference Guide](#) to familiarize yourself with the new processes under Kodo 3 and the new features available to you.

Also, be sure to read the [release notes](#) for all Kodo 3 releases, including the betas. The Notable Changes section of each note lists topics you should be aware of in your migration.

## D.2. Migrating from Kodo 3.0 to Kodo 3.1

---

The following are points to consider when upgrading from Kodo 3.0.x to Kodo 3.1.0:

- Kodo now uses its own logging framework by default instead of the Commons Logging framework. To use the Commons Logging framework as in older versions of Kodo, set the `kodo.Log` property to `commons`. See [Chapter 3, Logging \[161\]](#) for details.
- The `DBDictionary` now supports the `DefaultSchemaName` parameter. See [Section 4.3, “Database Support” \[169\]](#). The default is `null`, which will cause Kodo to check all schemas for unqualified tables. Setting this will cause Kodo to only check the specified schema for unqualified tables. `kodo.jdbc.Schemas` is now only used in conjunction with reverse-mapping processes to identify which schemas to examine for reverse mapping.

## D.3. Migrating from Kodo 3.1 to Kodo 3.2

---

The following are points to consider when upgrading from Kodo 3.1.x to Kodo 3.2.0:

- Configuration of the management and profiling capabilities has been simplified. All configuration now happens through the `kodo.ManagementConfiguration` property. For more information, please see [Chapter 12, Management and Monitoring \[309\]](#) and [Section 14.7, “Profiling” \[342\]](#).

---

# Appendix E. Implementation Notes

This appendix discusses implementation-specific details that may be important to developers.

## E.1. jdoFlags 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 you modify a default fetch group field in a `PersistenceCapable` object that is involved in a transaction, Kodo JDO sets 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.

---

# Appendix F. DataCache Integrations

Part of the Kodo JDO Performance Pack includes the datastore cache. This appendix discusses integrations of the Kodo datastore cache to third party caching solutions.

## F.1. Tangosol Integration

---

The Kodo JDO data cache can integrate with Tangosol's Coherence caching system. To use the Tangosol integration, set the `kodo.DataCache` configuration property to `tangosol`, with the appropriate plugin properties for your Tangosol setup. For example:

### *Example F.1. Tangosol Cache Configuration*

```
kodo.DataCache: tangosol(TangosolCacheName=kodo)
```

The Tangosol cache understands the following properties:

- `TangosolCacheName`: The name of the Tangosol Coherence cache to use. Defaults to `kodo`.
- `TangosolCacheType`: The type of Tangosol Coherence cache to use (optional). Valid values are `named`, `distributed`, or `replicated`. Defaults to `named`, which means that the cache is looked up via the `com.tangosol.net.CacheFactory.getCache(String)` method. This method looks up the cache by name as defined in the Coherence configuration.

### Note

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

## F.2. GemStone Gemfire Integration

---

The Kodo JDO data cache can integrate with GemStone's GemFire caching system. To use GemFire in Kodo you will need to change your `gemfire.properties` to have the property `enable-shared-memory=true`. You will also need to add both Kodo and GemFire to your classpath and then start a GemFire server.

```
prompt> gemfire start
```

By default, the GemFire data cache will use a GemFire region of `root/kodo-data-cache` and the GemFire query cache will use a region of `root/kodo-query-cache`. This can be changed by setting the optional property `GemfireCacheName`.

### *Example F.2. GemFire Cache Configuration*

```
kodo.DataCache: gemfire(GemfireCacheName=/root/My-kodo-data-cache)
kodo.QueryCache: gemfire(GemfireCacheName=/root/My-kodo-query-cache)
```

cache.xml

```
...
<shared-root-region name="root">
  <region-attributes>
    ...
  </region-attributes>
  <region name="My-kodo-data-cache">
    <region-attributes>
      </region-attributes>
    </region>
    <region name="My-kodo-query-cache">
      <region-attributes>
        </region-attributes>
      </region>
    </shared-root-region>
  ...
```

If you set GemFire for both `kodo.DataCache` and `kodo.QueryCache` you aren't required to specify a `kodo.RemoteCommitProvider` unless you are registering your own `RemoteCommitListeners`.

Some notes regarding using GemFire with Kodo:

- Your Application ID classes must be serializable (this is a JDO requirement also).
- Your classes used in fields that have externalizers or custom field mappings must also be serializable.
- The Kodo option of **`kodo.DynamicDataStructs`** isn't currently supported with the GemFire integration.

---

# Appendix G. 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-wl81manage.jar.jar`: Library required to connect to WebLogic 8.1 with `remotejmxtool`. This jar must be included in the WebLogic system classpath.
- `kodo-workbench.jar`: Launcher for the Kodo Development Workbench. This jar is never required for development or runtime; it exists solely for the purpose of being able to conveniently launch the workbench.
- `jcommon-0.9.1.jar`: Library of classes used by JFreeChart. Required when running the Kodo Management Console. See <http://www.jfree.org/jcommon/>.
- `jca1.0.jar`: Java Connector Architecture, used internally by Kodo. Required for all development and runtime use of Kodo. See <http://java.sun.com/j2ee/connector/>.
- `jdbc-hsql-1.7.0.jar`: Hypersonic pure-java database engine. It is used as a simple JDBC driver for learning Kodo, but it is not required for general development or runtime use. See <http://hsqldb.sourceforge.net/>.
- `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 being used (since the library is included with JDK 1.4). See <http://java.sun.com/products/jdbc/>.
- `jline.jar`: Console handling utility. Used by the `sqlline.jar` file. This file is only used for SQL debugging; it never needs to be included for Kodo runtime operations. See [Section 8.4, “The SQLLine Utility” \[284\]](#) for more details. Distributed from <http://jline.sourceforge.net>.
- `jdo-1.0.2.jar`: Interfaces defined by the Java Data Objects standard. Required for all development and runtime use.
- `jfreechart-0.9.16.jar`: Charting library used by Kodo Management Console. Required when running the Kodo Management Console. See <http://www.jfree.org/jfreechart/index.html>.
- `jndi.jar`: JNDI interfaces. Required for all development and runtime use of Kodo with JNDI, unless JDK 1.4 or higher is being used. See <http://java.sun.com/products/jndi/>.
- `jta-spec1_0_1.jar`: JTA interfaces. Required all development and runtime use of Kodo. See <http://java.sun.com/products/jta/>.
- `mx4j-admb.jar`: Library of JMX helper classes used by Kodo. Required for all development and runtime use. Part of the MX4J open source JMX implementation. See <http://mx4j.sourceforge.net/>.
- `mx4j-jmx.jar`: MX4J library. MX4J is an open source implementation of the JMX specification. It is the default JMX implementation used by Kodo. Required for all development and runtime use when no other JMX implementation is available. Note that when using another JMX implementation, this library should not be included. See <http://mx4j.sourceforge.net/>.
- `mx4j-tools.jar`: MX4J library. MX4J is an open source implementation of the JMX specification. It is the default JMX implementation used by Kodo. Required for all development and runtime use when no other JMX implementation is available. Note that when using another JMX implementation, this library should not be included. See <http://mx4j.sourceforge.net/>.
- `sqlline.jar`: Command line utility for executing SQL commands directly against a database. This file is only used for SQL debugging; it never needs to be included for Kodo runtime operations. See [Section 8.4, “The SQLLine Utility” \[284\]](#).

- `xalan.jar`: **Apache** implementation of the transformation parts of the Java API for XML Parsing. Any JAXP-compliant implementation can be used in place of `xalan.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 transformer. See <http://xml.apache.org/xalan-j/>.
- `xercesImpl.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/>.
- `xml-apis.jar`: XML interfaces. Required for Kodo development and runtime, unless JDK 1.4 or higher is being used.

---

# Appendix H. Release Notes

## 3.3.5 - TDB

- New features
  - Kodo management capability now supported in JBoss 4.

### Bugfixes

- The management function no longer directly references classes found only in the `kodo-jdo.jar`. This caused `java.lang.NoClassDefFoundErrors` in some configurations. Bug 1197.

## 3.3.4 - July 22, 2005

- Notable Changes
  - Pessimistic locks on embedded persistent types are now obtained at the same time as locks on their owning records (and vice versa), reducing the number of database roundtrips required to lock an instance and its embedded record(s).
  - Improved performance of getting `PersistenceManagers` when the `kodo.PersistentClasses` property is set.
- Bugfixes
  - The Kodo Workbench will be much faster when mounting many classes. Bug 1180.
  - The `SequenceGenerator.ensureCapacity()` method now uses the passed-in capacity number instead of the configured generator increment.
  - Fixed bug that cleared state of new instances that were not changed within the transaction when `RestoreValues` was true.
  - Removed extraneous `DISTINCT` from some eager fetching selects.
  - When using a third-party data source that is integrated with a JTA transaction manager, calls to `KodoPersistenceManagerFactory.getPersistenceManager(boolean managed, int retainMode)` will now return a `PersistenceManager` that uses `ConnectionFactory2` instead of the primary (managed) connection factory.

## 3.3.3 - May 20, 2005

- New features

- Added `DriverDeserializesBlobs` property to the `MySQLDictionary`. Most MySQL drivers automatically deserialize BLOBs on calls to `ResultSet.getObject`, but some do not.
- Kodo JDO Enterprise Edition and the Kodo JDO Performance Pack now includes a cache plug-in that supports GemStone's GemFire cache products. See the [data cache integration documentation](#) for details.
- Notable Changes
  - `com.solarmetric.profile.ProfilingHelper` and assorted classes were renamed to `ExecutionContextNameProvider` to better reflect the role that this interface plays in the Kodo profiling framework.
  - Query cache now considers parameters when calculating query key hash code (parameters were always part of equals comparisons).
- Bugfixes
  - The Kodo Workbench now correctly uses any classpath elements (such as JDBC driver jars) that you add to the Workbench when opening a query browser.
  - Fields beginning with multiple underscore characters will have all leading underscores properly trimmed. Bug 1125.
  - Fixed query caching problem with queries that involve subqueries. Bug 1122.
  - Fixed query cache interaction with certain remote commit provider configurations. Bug 1130.
  - Fixed potential exception when detaching an application identity class with embedded fields in the configured fetch groups. Bug 1142.
  - Fixed problem where in memory JDOQL queries using `String.matches` might match a value when it should not. Bug 1143.
  - Corrected case where using `this instanceof X` in a query whose candidate class is horizontally mapped would not limit the results to class X. Bug 1123.

### ***3.3.2 - April 14, 2005***

- Bugfixes
  - Fixed problem with base-horizontal-vertical hierarchy when using application identity. Bug 1114.
  - Fixed problem with `ConcurrentModificationException` in query cache. Bug 1115.
  - Fixed problem with ref-constant non-standard joins when using a negative number. Bug 1116.
  - Fixed problem with deadlock in datastore cache. Bug 1120.

### ***3.3.1 - March 28, 2005***

- Notable Changes
  - Throwing a `JDOFatalException` from a `TransactionListener` now propagates the exception to the user. The exception will cause a rollback if it occurs during a commit.
- Bugfixes
  - Corrected an inefficiency when performing a query against an entire hierarchy of mapped persistence-capable classes, where the candidate class extended a horizontally-mapped type.
  - Fixed problem with timestamps reporting as unknown column type on Oracle 10 with certain combination of driver and database. Bug 1111.
  - Fixed bug in new raw SQL handling abilities of `Row` class.
  - Fixed bug preventing Kodo from reflecting on all but the first schema-qualified table in the schemas list.
  - Corrected possible `NullPointerException` when committing a remote `PersistenceManager` transaction containing newly-persisted application identity objects in `Set` or `Map` fields.
  - Fixed bug in profiler where processing events could result in a `NullPointerException`.
  - Added setting `UseClobs` to allow for MySQL to allow clob use on versions which handle this correctly. Defaults to false for compatibility. Bug 1109.

### 3.3.0 - March 15, 2005

- New features
  - Added preview support for **single field identity**, a JDO 2 feature. This allows for the use of application identity without writing object id classes when using a single primary key field.
  - Added lifecycle event listening framework as a JDO 2 preview feature. This currently requires importing `kodo.event` instead of `javax.jdo`. See the **Lifecycle Events** section for more details.
  - Added preview support for `PersistenceManager.getObjectById(Class cls, Object value)`. This convenience method retrieves instances based on oid, primary key value, and stringified oids. See **Section 10.2.2, “JDO 2 Preview Methods” [288]**.
  - Added ability to dynamically generate data structs used for datacache and remote use. Classes are dynamically created to avoid primitive wrappers, optimizing memory use and load/store performance. See the **kodo.DynamicDataStructs** configuration property for more details.
  - Added ability to have caches evict based on schedule. Kodo's default caches can now parse "cron" style scheduling strings to evict at granularity from minute to month. See **Section 14.3.2, “Kodo JDO Cache Usage” [329]**.
  - New `KodoPersistenceManager.preFlush` method runs pre-flush actions such as persistence-by-reachability, deletion of dereferenced dependent objects, instance callbacks, and inverse relationship management without flushing. See the **Javadoc**.
  - New metadata extensions allow definition of the JDBC type or SQL type name for single column mappings (i.e. value mappings). See **Section 6.2.2.5, “jdbc-type” [201]** and **Section 6.2.2.6, “jdbc-sql-type” [201]**.

- Numeric fields can now have their value assigned from the sequence generator for the owning class. This is especially useful for assigning application identity primary key values. See [Section 6.2.4.9, “sequence-assigned” \[206\]](#).
  - New mapping tool argument allows tool to create SQL scripts rather than XML schema files or directly acting on the database. See [Section 7.1, “Mapping Tool” \[209\]](#).
  - Kodo now works with JMX 1.2 implementations, including those that implement JSR 160. Configuration of the management capability has changed slightly. See [Chapter 12, \*Management and Monitoring\* \[309\]](#).
  - Apache Derby is now a supported database. See [Section B.1, “Apache Derby” \[404\]](#).
  - Added support for `KodoPersistenceManager.refreshAll (JDOException)`. See [Section 10.2.2, “JDO 2 Preview Methods” \[288\]](#).
  - Enhanced `JMSRemoteCommitProvider` with options for passing properties to the JNDI `InitialContext` and for attempting to reconnect to the JMS topic if the JMS system notifies Kodo of a serious connection error. See [Section 14.4.1, “Remote Commit Provider Configuration” \[336\]](#).
  - Added ability to insert raw SQL into a given `kodo.jdbc.sql.Row`. This is useful for performing server side functions to generate values for custom field mappings.
  - Fields of type `Collection` and `Map` that use Java 5 generics no longer need JDO metadata specifying collection element type or map key / value types.
  - Java 5 enum field types can now be persisted natively (without the use of an externalizer). Such fields must still be listed in the JDO metadata file as `persistence-modifier="persistent"`, as enums are not among the spec-endorsed persistent field types in JDO1. Collections of enums and maps with enum keys or values are not yet supported. See [Section 7.9.8, “Enumeration Mapping” \[250\]](#) for details.
  - Kodo now provides a syntax for specifying additional lock groups in subclasses that are not used in the corresponding least-derived types. See [Section 14.6.1, “Lock Groups and Subclasses” \[341\]](#) for details.
- Notable Changes
    - Sequences assigned with the "db" and "db-class" sequence factories (see [Section 5.2.4.1, “Sequence Factory” \[185\]](#)) will now ensure that the assigned sequence value is at least 1 in order to be able to distinguish between auto-assigned values and cases where the default value is 0 for a primary key field.
    - Moved and repackaged Jakarta Commons and RegExp libraries internally. This removes the dependency on the Jakarta jars as well as avoids issues with conflicting library versions. Note that to use Commons Logging plugin instead of Kodo's default one, you still need to include the Commons Logging jar in your classpath.
    - The API of Kodo's internal datastore identity type, `kodo.util.Id` has been aligned with JDO 2 single field identity types.
    - Simplified and enhanced the datastore cache. Standard APIs like `DataCache.pin` and `DataCache.remove` are the same, but some less-used APIs have changed. See the Javadoc for the `kodo.datacache` package for details.
    - Changed the handling of object id assignment. Now, `PersistenceManager.getObjectId(pc)` and `JDOHelper.getObjectId(pc)` always return the final object id of the given instance. If the instance is new and uses auto-increment columns for its identity, these methods will cause a flush so that the identity value(s) can be determined. If the instance uses application identity, you cannot change any primary key fields after retrieving the object id.
    - Removed the `kodo.AutoIncrementConstraints` configuration property. Kodo now determines whether it needs to perform auto increment tracking dynamically.
    - The `kodo.jdbc.Schemas` property is now used to limit the tables visible during runtime schema validation in addition

to its traditional use to limit schema reflection in Kodo command-line tools.

- Altered the behavior of the detach methods when the `RollbackOnly` flag is set to allow multiple attach/modify/detach/rollback cycles. See [Section 11.1.2, “Detach and Attach Behavior” \[298\]](#) for details.
- The Coherence datacache plug-in now performs named cache lookups using the `com.tangosol.net.CacheFactory.getCache(String)` method by default. This differs from earlier Kodo versions, in which the `CacheFactory.getDistributedCache(String)` method was used by default. This change simplifies the configuration of the integration with Coherence, and allows use of Coherence's near cache capabilities, which were not previously accessible via Kodo's out-of-the-box configuration.
- `StoreManager.newDataStoreId()` signature has been changed to optimize access to non-string types.
- Aligned single-string JDOQL with most recent version of JDO 2 draft. Kodo still supports its previous single-string grammar, but see [Section 11.9, “Single-String JDOQL” \[70\]](#) for the grammar you should use going forward.
- Bugfixes
  - Added "CrossJoinClause" (??? [171]), "InnerJoinClause" (??? [171]), "OuterJoinClause" (??? [171]), and "RequiresConditionForCrossJoin" (??? [171]), in order to allow the customization of join clauses. See the description for bug #1103.
  - Made persisting many new instances in a remote persistence manager more efficient. Bug 1023.
  - In-memory queries involving `java.util.Date` fields work again. Bug 1057.
  - Queries involving non-managed parameters now do not have a side effect of making the parameter transactional. Bug 1061.
  - Fixed problem where `kodo.rar` does not deploy under JBoss 4.0. Bug 1063.
  - Fixed bug with queries using range and DISTINCT on SQLServer via changes to DBDictionary. Bug 1080.
  - Aggregate queries are no longer issued FOR UPDATE.

### ***3.2.4 - January 6, 2005***

- Bugfixes
  - Fixed bug with unique queries and query cache interaction.
  - Fixed query cache bug in which querying uncommitted changes in one transaction could cause `JDOObjectNotFoundException` in another persistence manager.
  - Kodo can now connect to JBoss 3.2.5 via JMX.

### ***3.2.3 - November 18, 2004***

- New features

- Added `MaximizeBatchSize` configuration option to default `kodo.jdbc.UpdateManager` property. Defaults to `true`, indicating that Kodo should sort statements in order to optimize batch size when statement batching is on.
- Notable Changes
  - `kodo.util.ProxyCollection` and `kodo.util.ProxyMap` no longer implement `java.util.Collection` and `java.util.Map` (respectively). This allows people to implement their own `ProxyCollection` in JDK 1.5 without compiler errors relating to generics.
- Bugfixes
  - Enabled ability to force no class indicator during reverse mapping process by specifying "none" as `kodo.jdbc.ClassIndicator`.
  - When Kodo encounters errors while processing a registered persistent type for the first time, it can now log a warning and retry the registration later instead of throwing an error. See [Section 2.6.41, “kodo.RetryClassRegistration” \[154\]](#).

### ***3.2.2 - October 15, 2004***

- New features
  - Added ability to receive a callback when Kodo discovers an orphaned database key, with built-in options for logging a message, throwing an exception, or doing nothing. See [Section 10.9, “Orphaned Keys” \[296\]](#).
  - Added ability to control JBuilder logging verbosity.
- Bugfixes
  - Fixed problem where reverse mapping the same classes twice in the Kodo Workbench would have issues when reloading the classes.
  - Fixed problem where setting the persistence-modifier of a field to "none" in the Kodo Workbench would not be preserved.
  - Prevent the connection pool from thinking that it is out of connections after repeated failed attempts to connect.
  - Fixed a bug in which conditions limiting a SELECT to certain subclasses could be left out when using final subclasses or the `kodo.PersistentClasses` property.
  - Fixed a bug keeping collection and map fields from being detached during detach-on-close.
  - Changed class-criteria constrained one to many fields to not null all back references when there was no inverse owner. The implicit inverse needs to be manually set to null in these cases.

### ***3.2.1 - October 5, 2004***

- Notable Changes
  - Changed the semantics of the `javax.jdo.option.RestoreValues` and `javax.jdo.option.RetainValues` properties to match JDO 1.0.1 specification. Previously, rollbacks with `RestoreValues` set but `RetainValues` unset would preserve the dirty state of the instances. They now transition to hollow. Conversely, previous rollbacks with `RestoreValues` unset but `RetainValues` set would transition the instances to hollow. They now rollback their state and transition to persistent-nontransactional.
  - Management and profiling of Kodo within WebLogic 8.1 is now supported. See **Chapter 12, *Management and Monitoring* [309]**.
- Bugfixes
  - Fixed bug that could allow relations to get out of synch in the datastore cache when managed relations were enabled.

### ***3.2.0 -- September 29, 2004***

- New features
  - The reverse mapping tool interface in the workbench is now a guided wizard that allows the user to customize various aspects of the reverse mapping process.
  - The workbench will now try to compile java files into classes when the source is available but the class file is not.
  - The reverse mapping tool can now generate inner classes for application identity classes with the `innerAppId` option. See **Section 5.6, “Auto-Generating Classes from a Schema” [193]**.
  - Added an optional `DiagnosticContext` to Kodo's logging implementation. If set, all log lines from the configured `PersistenceManagerFactory` will be prefixed with the token. See **Section 3.2, “Kodo Logging” [162]**.
  - Added support for single-string JDOQL queries, a proposed JDO 2 feature. See **Section 11.9, “Single-String JDOQL” [70]**.
  - Added support for implicit JDOQL parameters and variables, a proposed JDO 2 feature. See **Section 11.3, “Advanced Object Filtering” [58]**.
  - Added preview of JDO 2.0 named query support. See **Section 11.10, “Named Queries” [72]** and **Section 12.4, “Named SQL Queries” [79]**.
  - Support for subqueries in JDOQL. See **Section 13.3, “JDOQL Subqueries” [323]**.
  - Added optional automatic management of inverse relations. See **Section 5.3, “Managed Inverses” [188]**.
  - Added `KodoPersistenceManager.checkConsistency` to check the consistency of the persistence manager cache without enlisting additional database resources. JDO 2 preview feature.
  - Added new sample models to the `samples/` directory: **Section 1.15, “Sample Human Resources Model” [373]** and **Section 1.16, “Sample School Schedule Model” [373]**.
  - Added `ExceptionAction` connection pool property to determine what to do when connections that have thrown exceptions are returned to the pool. Previous versions of Kodo always destroyed these connections, and that is still the default action. See **Section 4.1, “Using the Kodo JDO DataSource” [166]**.

- Many improvements to the workbench, including the ability to preview the SQL DDL for classes in the workbench, the ability to print components, single-click workbench starting, the ability to dynamically edit the classpath, and many user interface tweaks.
- Added ability to auto-externalize constant simple values (primitives, primitive wrappers and Strings) through metadata extensions. See [Section 7.9.24, “External Values” \[276\]](#).
- You can now configure Kodo to detach objects with their currently-loaded fields or to detach all fields rather than detaching based on the current fetch groups. See [Section 11.1.3, “Defining the Detached Object Graph” \[299\]](#).
- Added ability to automatically detach objects when the persistence manager closes, or when they are serialized. See [Section 11.1.5, “Automatic Detachment” \[301\]](#).
- Vertically-mapped inheritance hierarchies no longer require a class indicator column. See [Section 7.8.3, “Subclass-Join Indicator” \[235\]](#).
- You can now configure Kodo to outer-join to vertically mapped subclass tables when fetching data, on either a global or class-by-class basis. See [Section 14.2, “Eager Fetching” \[326\]](#).
- Improved eager fetching. Multiple relation fields of the same type can now be eager-fetched at once. Eager fetching using parallel selects now respects large result set settings, such that the related objects are selected for each "page" of objects brought into memory. Ability to specify that certain collection fields should be eager-fetched with joins when possible, rather than with parallel selects. See [Section 14.2, “Eager Fetching” \[326\]](#).
- Custom JDOQL extension methods can now take multiple arguments.
- Kodo now supports InterSystems Cache. See [Section B.9, “InterSystems Cache” \[407\]](#).
- Added support for the `javax.jdo.query.SQL` query language introduced in the JDO 2 early draft specification. The `kodo.jdbc.SQL` query language is considered deprecated. See [Chapter 12, \*SQL Queries\* \[76\]](#) for details.
- SQL queries now support projections and custom result classes. See [Section 12.3, “SQL Projections” \[78\]](#).
- Support for grouping and having clauses in JDOQL queries, matching JDO 2 early draft specification. See [Section 11.7, “Aggregates” \[66\]](#).
- Support for setting result ranges on queries, matching JDO 2 early draft specification. Eager fetching works intelligently with query ranges so that the range is not affected by eager fetching and only eager data for the requested range is selected. See [Section 11.5, “Limits and Ordering” \[62\]](#).
- As per the JDO 2 early draft specification, JDOQL now supports the following new operators and functions: `instanceof`, `String.substring`, `String.indexOf`, `Math.abs`, `Math.sqrt`, `JDOHelper.getObjectId`. See [Section 11.2, “JDOQL” \[56\]](#).  
  
Additionally, many JDOQL functions previously supported as Kodo query extensions are now official parts of JDOQL: `String.toLowerCase`, `String.toUpperCase`, `String.matches`, `Map.containsKey`, `Map.containsValue`.
- Allow access to public static fields in JDOQL, matching JDO 2 early draft specification. See [Section 11.2, “JDOQL” \[56\]](#).
- Allow single-quoted string literals in query filters, matching JDO 2 early draft specification. See [Section 11.2, “JDOQL” \[56\]](#).
- Support for distinct keyword in query result string, matching JDO 2 early draft specification. See [Section 11.6, “Projections” \[64\]](#).
- Default query result string to `distinct this as C`, where `C` is the unqualified candidate class name. Matches JDO 2 early draft specification. See [Section 11.8, “Result Class” \[68\]](#).

- Support for setting public fields of query result classes in addition to setter methods, matching JDO 2 early draft specification. See [Section 11.8, “Result Class” \[68\]](#).
  - Added `Query.Extensions` map introduced in JDO 2 early draft specification.
  - Queries executed through the JDO query facilities are now logged to the `kodo.Query` channel. See [Chapter 3, Logging \[161\]](#).
  - Usage statistics for query results and collection and map type relations can now be viewed in the profiling tool. See [Section 14.7, “Profiling” \[342\]](#).
- Notable Changes
- The `KodoPersistenceManager.flush` method now returns silently if no transaction is active, rather than throwing an exception. Complies with the JDO 2 early draft specification.
  - Removed the `kodo.jdbc.VerticalQueryMode` configuration property and associated runtime APIs. Also removed the `kodo.jdbc.JoinSubclasses` property and metadata extension introduced in 3.2.0b1. Use the new consolidated `kodo.SubclassFetchMode` property. See [Section 14.2, “Eager Fetching” \[326\]](#).
  - Upgraded Apache Commons Collections and Apache Commons Pool versions that ship with Kodo.
  - Kodo now includes the release candidate of the JDO 1.0.2 jar (`jdo-1.0.2.jar`) instead of the JDO 1.0.1 jar. This fixes an internationalization bug in the JDO 1.0.1 jar. It is no longer necessary to use the `il8nhelper_websphere_patch.jar` patch.
  - The JDO jar (now `jdo-1.0.2.jar`) file is no longer included in the resource adapter file (`kodo.rar`). The JDO jar file should be deployed in the global classpath for the application server.
  - Changed `MaxCharactersInAutoIncrement` property of `DBDictionary` to `MaxAutoIncrementNameLength` to match naming conventions of other length restriction properties.
  - `kodo.datacache.TangosolCache` now uses the `CacheFactory.getCache()` method if the `TangosolCacheType` property is left unset. As a result, the return value of `TangosolCache.getDistributedCache()` has been deprecated, and will not return accurate information if the `TangosolCacheType` configuration property is not set.
  - Optimistic transactions in which no changes have taken place will no longer obtain and commit a datastore connection when the JDO transaction is committed.
  - Simplified the `DBDictionary` class for easier extension and greater configurability. The changes may break existing custom dictionaries. The source code for the new dictionaries is included in your Kodo distribution to help you migrate your custom dictionaries.
  - The query improvements in this release required changes to the `FilterListener` interface, and its `JDBCFilterListener` subclass. The source code for Kodo's built-in listeners is included in your Kodo distribution to help you migrate your custom listeners.
  - The query improvements in this release also required minor changes to some **FieldMappings**. See the Javadoc and the samples in `samples/ormapping` for details.
  - `kodo.jdbc.SQL` queries have been deprecated. Use JDO 2 preview SQL queries instead. See [Chapter 12, SQL Queries \[76\]](#).
  - The `FetchConfiguration.EAGER_FETCH_*` constants have been deprecated in favor of constants that more accurately reflect the semantics of each fetch mode. See [Section 14.2, “Eager Fetching” \[326\]](#).

- Attempting to query on an interface or abstract class without any persistent implementors will throw an exception.
- Configuration of the management and profiling capabilities has been simplified. For more information, please see **Chapter 12, *Management and Monitoring* [309]** and **Section 14.7, “Profiling” [342]**.
- Bugfixes
  - Query parameter validation now ensures that you do not pass extra parameters to a `Query.execute()` invocation. This may cause some of your previously-functioning queries to fail to execute.
  - Corrected Empress database dictionary to use `TOLOWER` and `TOUPPER` SQL functions. Bug 964.
  - Removed restriction limiting arguments to the `startsWith` and `endsWith` JDOQL methods to literals and parameters. Bug 970.
  - Fixed possible exception when attaching new embedded objects.
  - Worked around JDO library bug with potential infinite recursion when printing out stack traces that contain a failed object whose `toString` method accesses persistent fields. Bug 979.
  - Fixed exception when supplying a null value for an implicit parameter.
  - Query caching behaves properly with mutable parameter types (`Collection`, `Date`) that are changed between executions (bug 959)
  - Queries that return no results are properly cached.
  - Projections can now contain the same column multiple times; bug 853.
  - Added support for non-persistence capable query variables; bug 720.
  - All return types from projections and aggregates, whether executed in-memory or in the data store, now exactly match the types specified by the JDO 2 early draft specification. Bug 721.
  - Fixed some cases of not being able to use variables in projection and aggregate queries executed in-memory. Bug 713.
  - Fixed bug that prevented a `lock-group` of `none` from working in some cases.
  - Fixed a bug that prevented the reverse mapping tool from recognizing the `-s` cmd-line argument shortcut for the `-schemas` option.

### ***3.1.5 -- August 11, 2004***

- Notable Changes
  - When setting `kodo.ProfilingInterface` to `export`, default export interval is now `-1`, indicating that only a final export will be created.
- Bugfixes
  - Fixed possible exception when attaching a new embedded object.

- Included check for lock-group when using state-image version indicator.
- Final profiling export when setting `kodo.ProfilingInterface` to `export` is now produced.
- Fixed `DBDictionary` to allow column auto-increment sequence names longer than 31 by adding `MaxCharactersInAutoIncrement` property.

### **3.1.4 -- July 9, 2004**

- New features
  - `ClassDBSequenceFactory` can now ignore horizontally mapped classes when determining primary key values.
  - Firebird is now a supported database. See [Section B.7, “Firebird” \[407\]](#).
  - Borland Interbase is now a supported database. See [Section B.2, “Borland Interbase” \[405\]](#).
- Bugfixes
  - Corrected bug in `appidtool` that could result in invalid application identity subclass generation when the application identity superclass had one or more `Date` primary key fields.
  - The reverse mapping tool now honors the `DBDictionary`'s `UseSchemaName` property when deciding whether to qualify table names in generated mappings.
  - Fixed recently-introduced `NullPointerException` in some one-many mappings.

### **3.1.3 -- June 21, 2004**

- New features
  - Subclasses in an application inheritance hierarchy can now define additional primary key fields. See [Section 4.5.2.1, “Application Identity Hierarchies” \[25\]](#).
  - The Kodo Development Workbench now includes options for previewing the metadata and mappings in XML form. In addition, you can now edit foreign keys and use them in the visualization process.
  - The reverse mapping tool accepts a new option `-- -blobAsObject/-bo --` to map binary columns to `java.lang.Object` fields, rather than to `byte[]` fields. Enabling the option mirrors the tool's behavior in Kodo versions prior to 3.1.2.
  - Kodo will now issue warnings to the log when a mapping contains unrecognized attributes, or when a metadata extension contains an unrecognized key. The log message will include suggestions for similarly name attributes.
  - Kodo will now issue warnings to the log when an unrecognized property name is specified in the `kodo.properties` configuration file. The log message will include suggestions for similarly named properties.

- Notable Changes
  - The `AbstractStoreManager.newInstance` method no longer exists; use the `KodoStateManager.initialize(Class, JDState)` method in its place.
- Bugfixes
  - Fixed bug with id class validation problems with NetBeans plugin.
  - Fixed bug that could cause errors after rollback of a transaction in which an object with dependent relations was deleted.
  - Fixed externalization bug that could lead to a `ClassCastException` when one container type was externalized to another container type.
  - Fixed bug with caching of queries that use empty `Collection` parameters (bug 944).
  - Fixed bug in which reverse mapping tool generated invalid Java code for binary columns.
  - Changed DB2 SQL generation to not use `CROSS JOIN`, which DB2 does not understand.
  - Fixed bug that prevented Workbench from allowing you to edit metadata for field that are non-persistent by default.
  - Fixed query bug that could result in a `NullPointerException` when attempting to constrain a variable to a collection or map passed in as a parameter (or traversed from a persistence-capable parameter).
  - Fixed mapping bug when mapping an embedded field of a horizontally-mapped superclass to a subclass table.
  - Corrected an unnecessary `SELECT DISTINCT JDOCLASS` when using `PersistentClasses` list.

### ***3.1.2 -- May 21, 2004***

- Bugfixes
  - Fixed 3.1.1 bug that prevented use under some licenses.

### ***3.1.1 -- May 20, 2004***

- New features
  - Support for embedded one-to-one mappings in Kodo Development Workbench.
  - Support for data compression and filtering when using remote persistence managers. See **Section 11.2.4, “Data Compression and Filtering” [307]**.
  - Support for byte array primary key fields for legacy schemas that use binary columns for primary keys.
  - XML Store Manager sample now included. See **Section 1.14, “XML Store Manager” [372]**.

- The Kodo workbench now includes a live JDOQL query executor and result browser. See **Chapter 8, JDOQL Editor [399]**.
- Notable Changes
  - Changed default storage directory for Kodo Development Workbench from the current working directory to `${user.home}/solarmetric`. Kodo Development Workbench will handle the migration for you. However, you can choose to override this behavior by using the `-s` argument to specify a different location.
  - Byte array fields are now mapped using the byte array field mapping by default, rather than the blob field mapping. The byte array mapping does not serialize its data, while the blob mapping does. This should not affect existing mappings.
  - The reverse mapping tool now maps binary columns to `byte[]` fields, rather than `Object` fields.
  - Downgraded non-locking SELECT log messages from WARN to INFO.
  - Integration of `remotejmxtool` with unsupported application servers can now be plugged in. See **Section 12.2.1, “Remote Connection” [313]** for details.
  - Changed `kodo.runtime.LockManager` plugin API to allow the lock manager to access the connection info we are loading from, if any. This is useful for plugins that implement the `kodo.jdbc.runtime.JDBCLOCKManager` interface.
- Bugfixes
  - Changed class loading in Kodo tools so that static initializers of user-defined classes are not invoked in dev-time operation.
  - Fixed a possible constraint violation when using maps with restrict-action foreign keys to persistent object values. The error occurred when changing an existing map key and deleting the old value object.
  - Fixed a bug that could result in mapping errors when using numeric constant joins in combination with the dynamic schema factory.
  - Fixed a bug that could result in mappings to be generated for the implicit `"jdoDetachedObjectId"` field that is created when enhancing a detachable class. By default, any field starting with `"jdo"` will no longer be considered persistent unless it is explicitly declared.
  - Corrected an inefficiency in object locking that could result in a SELECT FOR UPDATE being issued for an already-locked object.
  - Fixed a bug that sometimes prevented Kodo from obtaining a requested database lock if the object was found in the data cache.
  - Removed potential deadlock in `JDBCConfiguration`.
  - Fixed bug that caused spurious graphical glitches and exceptions in JMX Management Console.
  - Removed a limitation that prevented non-serializable persistent instances from being used as parameters in a remote query. Note that this limitation is still in place for non-persistent instances.
  - Fixed bug that caused certain custom field mappings using `java.sql.Date/Timestamp` to get a `ClassCastException`.
  - Enabled App Id Tool to generate object identity classes for horizontally mapped classes.

- Fixed `ColumnVersionIndicator` to throw correct lock exception on certain conditions with deleted rows.

### 3.1.0 -- April 23, 2004

- New features
  - Added new configuration properties and runtime APIs for enhanced control over object locking. See [Section 10.8, “Object Locking” \[292\]](#) for details.
  - Kodo Development Workbench includes a number of new features and changes, including the ability to edit version and class indicators in the visualization editor.
  - You can now configure Kodo to evict objects from the data cache whenever you evict them from a persistence manager. See [Example 14.10, “Data Cache Eviction Through the Persistence Manager” \[332\]](#).
  - Added methods to `KodoHelper` to retrieve the data cache for an object or class.
  - Added example XML based `AbstractStoreManager` called `kodo.xmlstore.XMLStoreManager`.
  - Added support for large result set handling for fields declared as type `java.util.Set`.
  - Added support for constant joins in many-to-many relations, including mapping both a one-to-one and a many-to-many into the same join table.
- Notable Changes
  - Object locking APIs have changed. See [Section 10.8, “Object Locking” \[292\]](#) for details.
  - Added support for interface mappings in JBuilder and SunONE/NetBeans plugins in addition to other usability and stability fixes. Users of these plugins should install the new versions.
  - `ClassDBSequenceFactory` now has a `main (String[] args)` method to create and drop required schema components.
  - The `DBDictionary` now supports the `DefaultSchemaName` parameter. See [Section 4.3, “Database Support” \[169\]](#).
  - Optimized statement batching within groups of SQL updates ordered to meet foreign key constraints.
  - The `StoreManager` API has changed slightly in light of the object locking enhancements in this release. See its [Javadoc](#) for details.
- Bugfixes
  - Fixed a bug in which the new `lockPersistent` APIs could cause errors if used during an optimistic transaction.
  - Fixed SQL generated for removing columns on Empress.
  - Created special case for LOB handling when using a Weblogic datasource connecting to an Oracle database to circumvent the fact that Weblogic wraps the native Oracle LOB-handling classes.
  - Fixed bug in which passing duplicate oids to `KodoPersistenceManager.getObjectsById` could result in in-

ternal errors.

### ***3.1.0 RC2 -- March 24, 2004***

- New features
  - Added the ability to use remote persistence managers over HTTP/HTTPS. See **Chapter 11, *Remote and Offline JDO* [297]** for details.
  - Added an abstract store manager, which is a building block for allowing you to add support for non-relational data stores that Kodo does not support. See the **`kodo.abstractstore` javadocs** for documentation.
  - Added a **`SequenceGenerator.ensureCapacity(int count)`** method. Invoke this method to provide a hint to a sequence generator about how many times its `next()` method will be invoked.
  - Added support for Empress database.
- Notable Changes
  - Kodo Development Workbench has added some usability improvements in addition to a variety of bugfixes.
  - Eclipse plugin has now been changed to 2.1.0. This new version includes the ability to enhance and run Mapping Tool on multiple .jdo files at once. In addition, a more full range of Mapping Tool options such as `readSchema` and `ignoreErrors` are now available. The enhancer builder now delays til the end of building to avoid re-processing of metadata (and potential classloader issues related to this).

This version of the plugin includes also a number of bug fixes. It is recommended that you uninstall the old version of the plugin by completely removing the old `kodo.eclipse_2.0.0` directory and installing the new one.
  - Deprecated `SequenceGenerator.getNext()` in favor of `SequenceGenerator.next()`, which returns an unboxed long rather than a `java.lang.Number`.
  - Management capability now disabled by default. To find/create an MBeanServer and register MBeans set the `kodo.MBeanServerStrategy` configuration property. See **Chapter 12, *Management and Monitoring* [309]** for details.
- Bugfixes
  - Fixed bug that caused intermittent errors in remote persistence managers when the primary key values of an application identity instance were changed after it was made persistent, but before commit.
  - Improved JBuilder plugin stability. Bugs fixed include proper handling of cancel, drop, and file organization.

### ***3.1.0 RC1 -- March 10, 2004***

- New features

- Kodo now uses its own logging framework by default instead of the Commons Logging framework. To use the Commons Logging framework as in older versions of Kodo, set the `kodo.Log` property to `commons`. See [Chapter 3, Logging \[161\]](#) for details.
- **Kodo Workbench** now includes the ability to dynamically edit from the **Visualization** editor. Mapping Tool actions now work on the current instead of the saved versions of mapping information.
- The `kodo.jdbc.SynchronizeMappings` property (which was known as `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` in version of Kodo prior to 3.0) now allows Kodo to dynamically attempt to build O/R mappings and the database schema at runtime. See [Section 7.2, “Automatic Runtime Mapping” \[213\]](#).
- Notable Changes
  - Kodo's JCA connection factory can now be safely cast to a `KodoPersistenceManagerFactory` rather than just a `PersistenceManagerFactory`.
  - The reverse mapping tool now automatically strips illegal characters from table and column names when mapping them to Java identifiers. This is mostly a bug fix, but also affects users who have reverse customizer properties files renaming fields for which Kodo used to include invalid characters.
  - If the `DBDictionary.StoreCharsAsNumbers` property is set to `false`, `CHAR(1)` columns will be reverse-mapped to Java char fields rather than strings.
- Bugfixes
  - Fixed bug that caused Kodo to sometimes return an inaccurate size for query results when large result sets were enabled, the query involved to-many joins, and Kodo was configured to use `SELECT COUNT` to calculate the size.
  - Fixed bug preventing objects with null relations from being returned when those relations were eager-fetched and were to subclasses using vertical inheritance.
  - Fixed bug that caused a parse error when attempting to case with the full class name in a JDOQL filter string (i.e. `"((com.xyz.Foo) foo).bar == x"`). Bug 869.
  - Fixed bug in which fields declared in the class of the cast were sometimes not recognized when casting in JDOQL filter string. Bug 805.
  - The `SELECT COUNT` subselect issued when testing for a null 1-1 or empty collection/map in a query filter now uses the fully qualified table name (including schema).
  - Fixed `NullPointerException` when passing an array or collection containing null elements to certain persistence manager methods such as `retrieveAll`.
  - Fixed bug involving the customization of reverse-mapped field names. Bug 881.
  - Fixed reverse-mapping problem with Informix databases.
  - Fixed error when you add a new instance to an ordered one-many relation, flush, then extensively modify or re-order the elements of the relation, then commit. Bug 887.

**3.1.0b1 -- February 16, 2004**

- New features
  - Kodo now supports the "horizontal" O/R inheritance model, where each leaf of the inheritance hierarchy is mapped to a separate table. See [Section 7.6.4, “Horizontal Inheritance Mapping” \[224\]](#) for details.
  - Added the ability to use Kodo from remote client machines communicating with a persistence manager server. See [Section 11.2, “Remote Persistence Managers” \[303\]](#) for details.
  - Improved efficiency of the query compilation cache. After a query is compiled or executed once, subsequent compilations or executions of queries with the same properties (even in different persistence managers) are much faster.
  - Added custom lock group support, allowing field-level optimistic locking granularity. See [Section 14.6, “Lock Groups” \[340\]](#) for details.
  - Added object locking APIs to the `KodoPersistenceManager` for fine-grained control over object locking. You can also plug in your own locking scheme through the `kodo.LockManager` configuration property.
  - Added a `cancelAll` method to the `KodoPersistenceManager` that can be used to cancel any outstanding database statements issued by a `PersistenceManager`.
  - Added a new option for controlling how Kodo queries against class hierarchies that use vertical inheritance mapping. See [Section 7.6.3.3, “Vertical Select Modes” \[224\]](#) for details.
  - Significantly improved technology preview of Kodo Management / Monitoring capability based on JMX. Includes local and remote management functions, management of Datastore cache, Prepared Statement cache, and Kodo Datasource Connection Pool, and advanced performance analysis. See [Chapter 12, Management and Monitoring \[309\]](#) for details.
- Notable Changes
  - `kodo.jdbc.meta.ClassMapping` and `kodo.jdbc.meta.FieldMapping` are now interfaces. Custom implementations that previously extended these classes directly should now extend `kodo.jdbc.meta.AbstractClassMapping` and `kodo.jdbc.meta.AbstractFieldMapping`.
  - The `KodoQuery.FLUSH_*` constants have been deprecated in favor of equivalent constants in the `FetchConfiguration` interface for easier access.
- Bugfixes
  - Fixed bug with invoking `size()` on query results returned from SQL queries when using lazy result support.

### 3.0.3 -- February 20, 2004

- New features
  - Added a `ref-constant` attribute type to mapping data, so that you can perform constant joins (see [Section 7.5.2, “Non-Standard Joins” \[218\]](#)) on reference foreign keys, such as those used in one-many mappings.
  - Manually flushing the persistence manager before commit now releases all hard references to flushed dirty objects, allowing you to insert or update an unlimited number of objects within the same transaction without running out of memory.
  - Added the new `kodo.runtime.PreDetachCallback`, `kodo.runtime.PostDetachCallback`,

`kodo.runtime.PostAttachCallback` and `kodo.runtime.PreAttachCallback` interfaces that allow instances to be notified when they are being detached or attached. See [Section 11.1.4, “Detach and Attach Callbacks” \[301\]](#).

- Added support for simulating auto-increment fields under Oracle by using triggers. See [Section 4.3.2, “OracleDictionary parameters” \[175\]](#).

#### Notable Changes

- Deprecated the `KodoPersistenceManager.getState` method in favor of the new `KodoPersistenceManager.getStateManager` method, which takes in a persistence capable instance rather than an object id. The actual instance is required to make sure that the correct state is returned when multiple persistent-new application identity objects have the same object id. This is possible if they are persisted before their primary key fields are set to unique values and the transaction has not yet committed.
- The data cache now does not do any copying of `Locale` objects, as they are final and immutable. This means that if you cache an object with a `Locale` reference and then load that object from cache at a later time, the `Locale` will be identical (`==` will pass) in both objects.
- The `Mappings.getForeignKey` and `Mappings.setForeignKey` custom mapping helper methods now automatically suffix the given attribute prefix with `-column` (use null for an attribute of `column`).
- Bugfixes
  - Fixed bug that sometimes prevented changes in positions of ordered list elements from being committed to the database.
  - Stopped Kodo from occasionally iterating large result set fields on load and flush.
  - Fixed bug that left open result sets when using **large result set fields** of first class objects.
  - Fixed bug with caching of `Date` and mutable (proxied) custom SCO field types.
  - Fixed bug with improper lookup of unloaded related objects from cache (bug 836).
  - Fixed bug that caused eagerly-fetched to-many fields to sometimes not contain all elements if the field elements were constrained in the query being executed (bug 855).
  - Fixed bug that could cause exponential rise in commit time when committing many interrelated new objects.
  - Fixed bug that prevented multiple new instances from being able to be attached if they used application identity (bug 848).
  - Fixed bug that prevented attaching a graph that made a newly added instance persistent if it was reachable via multiple paths (bug 860).

### 3.0.2 -- January 24, 2004

- New features
  - Added a `TableTypes` property to the `DBDictionary` to allow configuration of the table types that will be considered when reflecting on the schema.

- Added a **jdbc-version-ind-indexed** and **jdbc-class-ind-indexed** class metadata extensions to control the indexing of version and class indicator columns, respectively.
- The DBDictionary now supports the `InitializationSQL`, `CatalogSeparator`, and `UseSchemaName` parameters. See [Section 4.3, “Database Support” \[169\]](#).
- Allow application identity classes to use custom sequence factories for their sequence generators.
- Notable Changes
  - Made version indicator columns indexed by default. The next time you run the mapping tool's `refresh` action, you may see indexes added to existing version indicator columns. To prevent this, set the new **jdbc-version-ind-indexed** class metadata extension to `false`.
  - Fixed bug that prevented the mapping tool's `buildSchema` action from adding indexes to the class and version indicator columns.
  - Changed the default `BatchLimit` DBDictionary setting for Oracle 9.2 Driver to handle statement batching issues. Users connecting using the recommended 9.0.1 driver are unaffected.
- Bugfixes
  - Fixed bug that caused invalid SQL when using the mapping tool's `buildSchema` action under Sybase.
  - Fixed bug that prevented the mapping tool's `buildSchema` action from adding indexes to the class and version indicator columns.
  - Fixed bug that caused invalid SQL when using large result set collection or map fields with types with compound primary keys.
  - Fixed bug that prevented inserts or updates of BLOBs over 4K in Oracle if the BLOB column had a NOT NULL constraint.
  - Fixed rare `NullPointerException` in query compilation cache.
  - Fixed problems with `SybaseDictionary`'s handling of `BigDecimal` and `BigInteger` values.
  - Fixed bug that sometimes led to duplicate objects in query results due to missing `DISTINCT` in database select.
  - Fixed issue with re-attaching detached instances with generic/unknown type fields.
  - Fixed metadata parsing issue with classes loaded under the bootstrap classloader in certain situations.
  - Fixed memory leak when invoking `Query.close()` (as opposed to `Query.closeAll()`) on a cached query.

### ***3.0.1 -- December 16, 2003***

- New features
  - Includes technology preview of the standalone Kodo Development Workbench. Kodo Development Workbench provides integrated mapping tools for your Kodo development, including metadata editors, visual relational graph analysis, config-

uration wizards, and access to development tools such as **SchemaTool** and **Reverse Mapping Tool**

- Added new **Byte Array Field Mapping**. This mapping avoids serialization of byte array fields for interactions with non-Java applications.
- The `sqlline.jar` utility is now included in the Kodo distribution. It is useful for a unified command interface for any database. See **Section 8.4, “The SQLLine Utility” [284]**, and for complete documentation, see <http://sqlline.sourceforge.net>.
- Added support for expressing nested O/R mapping extensions via XDoclet. See **Section 15.3, “XDoclet” [351]** for details.
- Introduced a cache for shareable query-related information. This improves query compilation times in many situations. See **Section 2.6.36, “kodo.QueryCompilationCache” [153]** for details.
- Added a `class-criteria` attribute to the **one-one mapping**.
- Added support for naming and configuring multiple data caches via the `kodo.DataCache` configuration property. See **Section 14.3.2, “Kodo JDO Cache Usage” [329]** for details.
- Notable Changes
  - Query extensions and aggregate queries are now included as part of Kodo Standard Edition; Enterprise Edition is no longer required to utilize these features.
  - Changed connection pool to reduce concurrency when creating and closing connections.
  - Changed the default value of the `kodo.RetainValuesInOptimistic` flag to `true` to comply with JDO 1.0.1 spec section 5.8. Note that this gives behavior similar to previous versions of Kodo; the change we made to the default behavior in Kodo 3.0 was incorrect.
  - Connection Decorators and JDBC listeners now do their work on all connections, inclusive of DBDictionary access.
  - Changed the handling of dependent fields to allow you to move dependent objects to other fields in a transaction. Kodo now only deletes dependent objects that have been removed from their owning field or had their owning object deleted, and that have not been assigned to any other field of any object. This analysis occurs on flush.
- Bugfixes
  - Fixed bug that prevented runtime licenses from working.
  - Optimized query compilation for datastore execution, not in-memory execution.
  - Fixed bug that inserted invalid rows into map tables when an existing key of a persistent map field was given a new value.
  - Fixed bug that caused direct SQL queries to throw an exception when executed in a transaction, even when no objects had been modified in the transaction and/or the global `javax.jdo.IgnoreCache` property was set to `true`.
  - Fixed bug that prevented the file mapping factory from working correctly when the path to metadata files contained spaces.
  - Fixed bug with Ant and Mapping Tool classloader conflict which caused `ClassNotFoundException`s.
  - Fixed bug 798 -- potential memory leak when query caching is enabled.

- Fixed a bug that caused inverse-based one-many and one-one mappings without an inverse-owner to sometimes produce bad SQL if the inverse columns were mapped to other fields as well, or were not nullable.

### ***3.0.0 -- November 7, 2003***

- New features
  - Kodo now supports direct SQL queries and stored procedure calls through the `javax.jdo.Query` interface. See [Section 13.4, “Direct SQL Execution” \[324\]](#) for details.
  - Technology preview of Kodo Management / Monitoring capability. See [Chapter 12, \*Management and Monitoring\* \[309\]](#) for details.
- Notable Changes
  - Added documentation for deploying in JRun 4.
  - The `ext:namedQuery` JDOQL extension has been replaced with the `kodo.MethodQL` query language. See [Section 13.5, “MethodQL” \[324\]](#) for details.
  - By default, the mapping tool no longer reads the entire existing schema on startup, though this option is still available. Also, the mapping tool does not examine or manipulate indexes, foreign keys, or primary keys on existing tables by default, though these options, too, are available as flags.
- Bugfixes
  - Fixed a bug that produced incorrect and sometimes invalid SQL when eager-fetching an inverse one-one relation.
  - Fixed a bug introduced in RC 3 that could cause collections and maps to be loaded as null whenever the datastore cache was enabled.
  - Fixes bugs in the IDE plugins. Please re-install any previous installations before installing the new versions of the plugin.

### ***3.0.0 RC4 -- October 31, 2003***

- Notable Changes
  - The `jdbc-use-*` metadata extensions have been renamed to `jdbc-*-name` extensions. So, for example, the `jdbc-use-class-map` extension is now `jdbc-class-map-name`. This change actually happened in RC3, but was not fully documented.

### ***3.0.0 RC3 -- October 31, 2003***

- New features
  - Added a `KodoHelper.getSequenceGenerator` method to ease obtaining a sequence for application identity classes.
  - Built-in support for Borland JDataStore, as well as Microsoft Access and Microsoft Visual FoxPro (using a JDBC-ODBC server bridge like DataDirect, but not the Sun JDBC-ODBC bridge).
- Bugfixes
  - Fixed synchronization problem that could result in a `ConcurrentModificationException` when Kodo is used under high load with managed transactions.
  - Fixed problem with incremental flushing that made it impossible to edit an existing object, flush, and then delete.

#### Notable changes

- The behavior of the `data-cache-timeout` metadata extension has changed considerably. In previous release candidates, a value of 0 meant that a class should never expire, and a value of -1 meant that the class should be excluded from the cache altogether. As of this version, a value of -1 means that the data in the cache should not expire, and is the default. 0 is no longer a valid value.

Additionally, the `data-cache-name` metadata extension's name has been changed to `data-cache`. Its role has been expanded to control disabling caching for a particular cache. To disable caching, set the `data-cache` extension to `false`. The default value for this extension is `true`.

So, to sum up, if you had a `data-cache-timeout` extension set to 0, you will get an exception when the metadata is parsed. Change the value to -1 instead. If you had set the extension to -1, then things are a bit trickier -- this will change the semantic behavior of your class unless you remove the extension and set the `data-cache` extension to `false`.

- Removed the `xa` option from the `kodo.TransactionMode` configuration property. When using an XA data source or other data source that is automatically involved in the global transaction, set `kodo.TransactionMode` to `managed` and set the new `kodo.jdbc.DataSourceMode` property to `enlisted`.
- The default value for `javax.jdo.option.IgnoreCache` has been changed from `false` to `true`.
- 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 `kodo.DataCacheConnects` property, as it is not needed now that locks are not obtained on the data cache.
- Re-worked SunONE/NetBeans and JBuilder plugins to remove a number of bugs as well as to improve the UI. Editors now incorporate commonly used Kodo extensions. For users of earlier versions, we recommend un-installing and re-installing the plugin.
- Made assorted minor tweaks to prepared statement and query caching.

### ***3.0.0 RC2 -- October 9, 2003***

- New features

- None
- Bugfixes
  - Fixed an optimistic locking error when the data cache is enabled.
  - Fixed some minor attach/detach bugs.
  - Fixed a bug in which persistent non-transactional objects were not cleared and reloaded when dirtied in an optimistic transaction. This could lead to false optimistic lock exceptions. See notable changes section below for details.
- Notable changes
  - Added the `kodo.DataCacheConnects` property to determine whether the data cache obtains a connection before each cache access. See the last list item in **Section 14.3.7, “Known Issues and Limitations” [335]** for details.
  - Kodo now clears and reloads persistent non-transactional objects when they are dirtied in an optimistic transaction. This is correct behavior as far as the spec is concerned, but can result in lower performance than the previous non-clearing behavior under certain usage patterns. Also, this change means that non-transactional writes can no longer be committed. Users who would like to continue with the old non-clearing behavior in order to increase performance or allow non-transactional writes to be committed should set the `kodo.RetainValuesInOptimistic` configuration property to `true`. Users of optimistic transactions and non-transactional reads who choose not to set this property should watch out for the following usage pattern:

```
Person p = (Person) pm.getObjectById (oid, false);
pm.currentTransaction ().begin ();
p.setName ("New Name"); // this now causes a reload of the object state!
pm.currentTransaction ().commit ();
```

If you are looking up data in order to modify it, make sure to look it up within the transaction rather than just before the transaction, and thus avoid a state refresh.

### ***3.0.0RC1 -- 23 September 2003***

- New features
  - Support for **detaching and attaching** instances from a persistence manager, allowing applications to more easily use a "data transfer object" pattern.
  - Support for collection and map fields that are **backed by large result sets**.
  - Support for additional settings to control the handling of **large result sets**.
  - Better exception messages when mappings can't be found or fail validations against the schema.
- Bugfixes

- Fixed many eager-fetching SQL errors.
- Notable changes
  - Kodo now uses non-scrolling result sets by default, and fetches all query results up-front by default. See the **large result set** section of the reference guide for how to configure large result set handling if needed.
  - The `JDBCQuery`'s `JoinSyntax` property has been moved into the query's `JDBCFetchConfiguration`. You should no longer cast to `JDBCQuery`, since that cast can fail when the data cache is enabled. Use the fetch configuration instead.

### ***3.0.0b2 -- 3 September 2003***

- New features
  - Expanded **smart proxies** to include change-tracking for lists.
  - Kodo now examines the initial field value of managed fields and stores any custom comparators for use when loading data into that field from the database.
  - Added the `kodo.RestoreMutableValues` configuration property.
  - **IDE plugins** have been re-implemented for Kodo 3. Support for NetBeans, SunONE Studio, JBuilder, and Eclipse have been re-added. Uninstall previous versions of the plugin, and follow the documentation for each plugin's installation.
  - New **externalization** system for storage of field types not supported by JDO without resorting to serializing or requiring custom mappings. Replaces the old stringification mapping.
  - Support for aggregates and projections in queries. See **Chapter 11, Query [54]** for details.
  - Added a `matches` query extension for limited regular-expression matching in JDOQL. See the **Included Query Extensions** documentation for details.
  - Documentation for how to use the latest builds of XDoclet to generate JDO metadata from source comments is now in the Integration chapter of the reference guide. A new XDoclet sample was also added to the `samples/` directory.
  - Added APIs for `get/setRollbackOnly` to the `KodoPersistenceManager` interface.
- Bugfixes
  - Fixed a bug in the first beta that prevented MySQL tables with VARCHAR columns from being created correctly.
  - Fixed a bug in the first beta that prevented eager-fetching from working efficiently. Also fixed some cases that could lead to stack overflow errors when using eager fetching.
  - Fixed a case in which compound primary keys could sometimes cause array index out of bounds errors when retrieving objects.
  - Fixed the Kodo 2 migrator tool, which sometimes specified arrays with the `<collection>` element in the migrated metadata.

- Improved support for columns shared by both relation foreign keys and simple value fields.
- Improved error messages when setting the same column to multiple different values or when trying to insert multiple objects with the same oid.
- Notable changes
  - Configuration properties specifying system plugins have been consolidated. See the **Plugin Configuration** section of the **Configuration** chapter for details. Beta 1 users may want to re-run the Kodo 2 properties migration tool on their old Kodo 2 properties instead of modifying their beta 1 properties by hand.
  - The default **mapping factory** has been changed to the file-based factory. If you were using the default database mapping factory in beta 1, either **switch to the file mapping factory**, or add the following line to your properties file:

```
kodo.jdbc.MappingFactory: db
```

- Changed the default table and column names for the DBSequenceFactory. If you were using the DB sequence factory in beta 1, either re-run the Kodo 2 properties migration tool, or add the following line to your properties:

```
kodo.jdbc.SequenceFactory: PrimaryKeyColumn=PKX, SequenceColumn=SEQUENCEX, \
    TableName=JDO_SEQUENCEX
```

- The stringification field mapping was replaced by the **externalization** framework.
- The `stringContains` and `wildcardMatch` query extensions have been deprecated in favor of the `matches` query extension.

### ***3.0.0b1 -- July 24, 2003***

- New features
  - New mapping system, providing more flexible mapping options. See the chapter on **Object-Relational Mapping** for more information.
  - Pluggable system for storing mapping information, with built-in options for storing mapping information in the database, in JDO metadata extensions, and in a separate mapping file. See the section on the **Mapping Factory** for more information.
  - Support for embedded 1-1 mappings, including nested embedded mappings with no limit on nesting depth. See the section on **Embedded One-to-One Mapping** for more information.
  - More complete mapping support for interfaces, including support for interfaces as the element type of collections, and the key and value types of maps. See the section on **Field Mapping** for more information.
  - Support for other-table mappings with outer joins. See the section on **Value Mapping** for more information.

- Support for 1-many fields without an inverse 1-1 field. See the example **Using a One-Sided One-to-Many Mapping** in the section on **One-to-Many Mappings** for more information.
- Support for first class objects as map keys. See the sections on **Many-to-N Map Mapping**, **Many-to-Many Map Mapping**, **PC Map Mapping**, **PC-to-N Map Mapping**, **PC-to-Many Map Mapping**, and **Many-to-PC Map Mapping** in the section on **Field Mapping** for more information.
- Support for non-standard joins, including partial primary key joins, non-primary key joins, and joins using constant values. See the section on **Non-Standard Joins** for more information.
- New samples for custom field mappings. The `samples/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.
- Support for automatically ordering SQL operations to meet all foreign key constraints, including circular constraints. See the section on **SQL Statement Ordering & Foreign Keys** for more information.
- Support for `javax.jdo.option.NullCollection`.
- Support for timestamp and state-based optimistic lock versioning, and for custom versioning systems. See the section on **Version Indicators** for more information.
- More configuration options for connection pooling. See the section on **kodo.ConnectionFactoryProperties** for more information.
- Support for SQL logging on third-party `javax.jdo.DataSources`.
- Configurable **eager fetching** of 1-1, 1-many and many-many relations. Potentially reduces the number of database queries required when iterating through an extent or query result and accessing relation fields of each instance.
- Ability to obtain both managed and unmanaged persistence managers from the same `PersistenceManagerFactory`.
- Support for auto-increment fields.
- Better support for auto-incrementing primary keys. See the section on **Primary Key Generation** for more information.
- More optimized **SQL batching**.
- Fail-fast error messages when object-relational mappings, class definitions, and schema are not in synch.
- Automatic schema generation now names schema components better, and automatically avoids naming conflicts with existing schema components.
- Simplified package structure and plug-in APIs. See the section on **JDO Runtime Extensions** for more information.
- Bugfixes
  - Fields in the same class hierarchy which share the same name no longer cause an invalid schema.
- Notable changes
  - A series of steps must be followed in order to migrate from Kodo 2.4 or Kodo 2.5 to Kodo 3.0. Please see **Appendix D: Migrating from Kodo 2 to Kodo 3** for more information.
  - The `schematool` has been replaced with the more powerful `mappingtool`. The `schematool` still exists, but now

has a different purpose. See the section on **Schema Tool** for more information.

- In Kodo 2.x, the default-fetch-group was not loaded when an object transitioned from hollow to a stateful state because a non-default-fetch-group field was loaded. Because `InstanceCallbacks.jdoPostLoad()` is invoked after the default-fetch-group is loaded, this meant that the `jdoPostLoad()` callback was not invoked in some circumstances when it might otherwise be expected to be invoked. kodo 3 always loads the default-fetch-group when an object transitions from hollow, so `jdoPostLoad()` will now be invoked in situations in which it was not invoked in the past.
- Beta Notes
  - Integration with the supported IDEs is not included in 3.0.0b1. IDE integration will be included in later distributions.
  - XDoclet integration is not included in 3.0.0b1. It will be added in a later release.
  - Support for DB2, Informix and Sybase is not included in 3.0.0b1. Support for these databases will be included in later distributions.
  - Enterprise integration is not fully tested in 3.0.0b1. Full testing and support for will be included in later distributions.

### ***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](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 J2EE 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:

```
com.solarmetric.kodo.DataCacheClass: com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl
com.solarmetric.kodo.RemoteCommitProviderClass: com.solarmetric.kodo.runtime.event.impl.JMSRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
```

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.SingleJVMRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
```

- 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 PreparedStatements and batch updates whenever possible for very significant performance benefits. See the documentation for the `com.solarmetric.kodo.impl.jdbc.UsePreparedStatements` and `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements` properties.
- Inverse one-to-one mappings are now supported. The field can now reside on either table corresponding to the related objects. If both sides of an one-to-one are marked as having an inverse, one field should be designated as read-only to indicate to the system the owning class and table for the given relational field.
- Logging is now done through the Apache commons project, which offers the ability to use an underlying logging mechanism, such as Apache Log4J, JDK 1.4's native logging, or simple file/stdout logging. It is now necessary to include the new `jakarta-commons-logging-1.0.2.jar` in the CLASSPATH. See the **Logging** chapter.
- Added a pluggable `SQLExecutionManager` architecture, which allows the developer to override the mechanism by which SQL is issued to the database. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass` property.
- There is now an option to automatically refresh the database schema during runtime, allowing the developer to skip the `schematool` step. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` property.
- Properties may be specified for a Driver with the `com.solarmetric.kodo.ConnectionProperties` property.
- A `javax.sql.DataSource` may be specified in the **`javax.jdo.option.ConnectionDriverName`** property, which will be customizable with bean-like entries in the `com.solarmetric.kodo.ConnectionProperties` property.
- The default transaction isolation for a JDBC connection can be overridden with the **`com.solarmetric.kodo.impl.jdbc.TransactionIsolation`** property.
- Kodo JDO now distributes a single jar for both the enterprise and standard edition, as well as datacache and query extensions.
- The `rd-metadatatool` can now be used to generate default JDO metadata for classes.
- The **`rd-schemagen`** tool used for reverse mapping classes from a schema has now been tested with the following databases: Hypersonic SQL 7.1, SQLServer (MS Beta 2 JDBC driver), Sybase, Oracle (9.0.1 JDBC driver), DB2, Postgres (7.3 Beta 3 JDBC driver).

#### Bugfixes

- `default-fetch-group="false"` is now respected for fields that default to `default-fetch-group="true"`.
- Traversing orphaned relations in data cache now behaves in the same way as traversing orphaned db relations -- the invalid relation is set to null.
- Changing a field to the same value as it was originally set to no longer constitutes dirtying that field. This means that subsequent flushes to the database will not necessarily re-write the same data.
- Class loading is performed in accordance with section 12.5 of the JDO specification.

#### Notable changes

- The API for implementing a `SequenceFactory` has changed. See the API documentation for **`com.solarmetric.kodo.impl.jdbc.SequenceFactory`**.

- Persistent types are no longer enumerated, either in the data store or in a property. Classes are now dynamically added upon class initialization, via the `JDOImplHelper` class registration process. The `-register` and `-unregister` options to `schematool` are no longer needed. The `JDO_SCHEMA_METADATA` table is no longer used and can be dropped.

### 2.3.4

- New features
  - The R&D schema generator can now accept a list of tables to generate.
  - The R&D reverse mapping tool has additional options for using foreign key names to generate relation names, generating primitive wrapper-type fields if a column is nullable, and allowing primary keys on many-to-many join tables.
- Bugfixes
  - Fixed problems with many-to-many relations between tables that use vertical inheritance.
  - Fixed bug in `schematool` that caused it to not generate primary key columns in subclass tables when using datastore identity + custom names + vertical inheritance.
  - Fixed `serp` library conflict between reverse mapping tool and main Kodo libraries.
  - Fixed a reverse mapping tool bug in which column names that conflicted with Java keywords would result in the generation of uncompileable Java classes.
  - Fixed problem that caused read-only flag to be ignored in many-to-many relations.
  - Multi-table inheritance deletes are now performed from the leaf table in the inheritance chain up to the base table. Inserts are performed from the base table down to the leaf. This supports the common referential integrity model of establishing a foreign key relation from inherited tables to their parent tables.

### 2.3.3

- New features
  - The R&D schema generator can now accept a list of tables to generate.
- Bugfixes
  - Fixed `null-value="default"` behavior.
  - Fixed bugs that prevented removal of map elements through the key set, entry set, and values collection.
  - Added more validation on static/final fields to metadata.
  - Fixed memory leak in `serp` regarding soft and weak collections backed by `HashSets`.

- Multi-variable query issues resolved.
- Fixed bug that could cause optimistic lock version numbers to be incremented before successful transaction commit.
- The R&D reverse mapping tool now handles Oracle DATE columns correctly.

### 2.3.2

- New features
  - The new `com.solarmetric.kodo.CacheReferenceSize` property dictates a number of hard references to cached objects that the `PersistenceManager` will retain, in addition to its soft cache.
  - Added 'all' option to unregister action of schematool Ant task. This option allows all classes in the current persistent types list to be unregistered, regardless of whether or not those classes are currently in the classpath.
  - Added `com.solarmetric.kodo.UseSoftTransactionCache` to configure whether or not Kodo should maintain soft references to transactional items that have not been dirtied. This now defaults to false; previous versions of Kodo always used soft references for non-dirty transactional items.
- Bugfixes
  - The `jdodoclet` task no longer creates JDO metadata entries for final or static fields, or for transient fields that do not have a `jdo:persistence-modifier` tag.
  - The `jdoc` task no longer attempts to enhance classes that have already been enhanced.
  - Several default property values were being set improperly.

### 2.3.1

- New features
  - Smart proxies for set and map fields. Smart proxies better optimize database updates when persistent set and map fields are modified.
  - TCP, JMS-based distributed `DataCache` implementations.
  - All `DataCache` implementations now use an LRU cache with a configurable maximum size.
  - Customizable tracked instance proxies.
  - Alpha release of upcoming reverse mapping tool for creating persistent class definitions, metadata, and mapping extensions from an existing schema.
  - Cache object `com.solarmetric.kodo.runtime.datacache.PMFactoryCache` is now named `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`.

- Bugfixes
  - A Query with an unspecified filter defaults to a filter of "true", rather than "false".
  - A Query with an unspecified candidate Extent but a specified candidate Class will automatically create an Extent of the appropriate type with subclasses turned on.
  - Various bugs related to compiled queries with null arguments or no parameters have been resolved.
  - Various InstanceCallbacks interface bugs have been resolved.
  - Ant schematool and jdoc tasks deal with the Ant ClassLoader system better.
- Notable changes
  - Made `GenericDictionary.toSQL()` and `GenericDictionary.fromSQL()` methods `final`. Subclasses of `GenericDictionary` that must change the behavior of SQL generation or parsing should do so by overriding the appropriate `xxxToSQL()` or `xxxFromSQL()` methods instead. Note that the source for all our dictionary classes is now available in the Kodo JDO distribution.

## 2.3.0

- New features
  - JDO specification version 1.0 support.
  - Highly flexible multiple-table inheritance model now supported. See the multi-table class mapping documentation for details.
  - Support for large result sets when using any JDBC 2.0+ driver that supports `ResultSets` of type `TYPE_SCROLL_INSENSITIVE`. Return values from all `Query.executeXXX()` methods will be an instance of `java.util.List`, which can then be used for efficient random access.
  - `DataCache` API batches distributed updates, facilitating custom processing of distributed cache invalidation.
  - `DataCache` implementation loads data read from the data store into the cache as well as data being written to the data store.
  - JDBC back-end customizability is improved, allowing for custom field and class mappings and much finer-grained control of generated SQL.
  - Kodo JDO now supports extending JDOQL with custom tags. A number of default extensions, including substring searches and case-insensitive searches, are included by default with Kodo JDO. For more on this feature, see the query extensions documentation.
  - Support for IBM WebSphere, and other application servers that do not provide a `TransactionManager` though 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 Database-MetaData.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 I. 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:

- Fields of an unknown type (i.e. `java.lang.Object` ) or a user-defined type that is not persistence-capable must be serializable.
- Queries do not support the `-` operator used for unary negation (it can be used for subtraction and to represent negative numbers).
- Using a cast in a Query made against an extent may not prevent instances of the base class from being returned.

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.

Please see the Kodo JDO [bug tracking database](#) for an up-to-date bug list.

---

# Index

## A

- ACID properties, 50
- Aggregates (see Query)
- Apache Ant, 346
  - application identity tool ant task, 348
  - configuration options, 346
  - mappingtool task, 349
  - metadata tool ant task, 349
  - Schema generator task, 350
  - schema tool task, 350
- appidtool, 24
- applets, 303
- Application Id Tool, 140
- Application identity, 23, 184, 228
  - ant task, 348
  - auto-generating app id classes, 24
- Atomicity, 50
- attach/detach, 297
  - automatic, 301
    - detach on close, 301
    - detach on serialize, 302
  - callbacks, 301
  - declaring, 297
  - graph definition, 299
  - jdoPostAttach, 301
  - jdoPostDetach, 301
  - jdoPreAttach, 301
  - jdoPreDetach, 301
- auto-increment, 187

## B

- Binary Compatible, 17
- blob mapping, 238
- buildSchema
  - (see also mappingtool)

## C

- caching
  - data cache, 329
  - Datastore Cache MBean, 318
- class indicator, 232, 236
- class maps, 219, 222
  - custom, 228
- CLASSPATH, 86
- clob mapping, 240
- connection pooling, 166
  - ExceptionAction, 166
  - MaxActive, 166
  - MaxCachedStatements, 168
  - MaxIdle, 166
  - MaxTotal, 166
  - MaxWait, 167
  - MinEvctableIdleTimeMillis, 167

- QueryTimeout, 168
- RollbackOnReturn, 167
- TestOnBorrow, 167
- TestOnReturn, 167
- TestWhileIdle, 167
- third-party datasources, 168
- TimeBetweenEvictionRunsMillis, 167
- ValidationSQL, 167
- ValidationTimeout, 167
- WhenExhaustedAction, 167
- ConnectionFactoryName, 38
- Consistency, 50
- Custom fetch groups (see Fetch groups)

## D

- DatabaseMetaData, 278
- datastore, 87
- Datastore identity, 22, 184
- DBDictionary, 169
- default-fetch-group attribute, 30
- delete action, 202
- deleting objects, 90
- Deploying Kodo in J2EE, 287
  - manually binding into JNDI, 108
- detach (see attach/detach)
- distributed inheritance mapping (see horizontal inheritance mapping)
- Durability, 50
- DynamicSchemaFactory, 279

## E

- Eager Fetching, 326
- eager fetching
  - fetch groups, 337
- EJBs (see Enterprise Java Beans)
- embedded, 30, 248
- embedded objects, 248
- Enterprise Java Beans, 13
- enum mapping, 250
- Extent, 53

## F

- fetch configuration, 290
- Fetch groups, 20, 30, 337
  - eager fetching, 326
- field mapping, 236
- Field Types
  - Immutable, 18
  - Mutable, 18
  - User-defined, 20
- field-level factory, 274
- filtered inheritance (see flat inheritance mapping)
- flat inheritance mapping, 220, 222

## G

- getCandidateClass, 53
- getObjectId, 33
- getPersistenceManager, 33, 37, 41
- getPersistenceManagerFactory, 36

**H**

hasSubClasses, 53  
horizontal inheritance mapping, 224, 228, 228

**I**

Identity  
    Application (see Application identity)  
    Datastore (see Datastore identity)  
    Single Field (see Single Field Identity)  
Immutable Field Types (see Field Types)  
in-class-name indicator, 232  
InstanceCallbacks, 20  
inventory maintenance, 88  
Isolation, 50

**J**

J2EE  
    deploying Kodo (see Deploying Kodo in J2EE)  
    Managed transactions, 321  
Java Database Connectivity, 12  
Java serialization, 11  
Java Serialization, 12  
Java Web Start applications, 303  
JCA, 103  
JDBC (see Java Database Connectivity)  
    connections, 177  
JDBCPersistenceManagerFactory, 103  
JDO 2.0 Features  
    attach (see attach/detach)  
    detach (see attach/detach)  
    queries, 54  
        aggregates, 66  
        grouping, 67  
        having, 67  
        JDOQL, 56  
        projections, 64  
        result class, 68  
        result range, 63  
        single-string, 68  
        SQL, 54  
        unique, 62  
JDO Enhancer, 16  
JDO Enhancers  
    Binary-Compatible (see Binary Compatible)  
JDO Exceptions, 15  
JDO Helper, 33  
JDO Identity Object, 21  
JDO Metadata, 27  
    DTD, 27  
    Placement, 31  
JDO runtime extensions, 288  
JDO Standard Properties, 141  
    ConnectionDriverName, 142  
    ConnectionFactory2Name, 142  
    ConnectionFactoryName, 142  
    ConnectionPassword, 142  
    ConnectionURL, 143

    ConnectionUserName, 143  
    IgnoreCache, 143  
    Multithreaded, 143  
    NontransactionalRead, 143  
    NontransactionalWrite, 144  
    Optimistic, 144  
    RestoreValues, 144  
    RetainValues, 144

jdoc, 86

JDOHelper, 33, 37  
    (see also KodoHelper)

jdoPostLoad (see InstanceCallbacks)

jdoPreClear (see InstanceCallbacks)

jdoPreDelete (see InstanceCallbacks)

jdoPreStore (see InstanceCallbacks)

JDOQL, 56, 88, 101

    (see also Query)

    aggregate extensions, 290

    extensions, 291

    subqueries, 323

JMX (see management)

JNDI

    manually binding Kodo into (see Deploying Kodo in J2EE, manu-  
    ally binding into JNDI)

JTA (see )

**K**

Kodo JDO

    configuration, 139

    datasource (see connection pooling)

    Performance Pack, 326

    plugin configuration, 140

    properties, 145

    runtime configuration, 139

    third-party datasources, 168

KodoExtent, 290

KodoHelper, 89, 290

    (see also JDOHelper)

KodoPersistenceManager, 288

KodoQuery, 290

**L**

Large Relations, 190

Large Result Set Proxies, 190

Large Result Sets, 180

lazy loading

    configuring relation loading, 326

    fetch groups, 337

    locking behavior, 295

LazySchemaFactory, 278

Lifecycle callbacks (see InstanceCallbacks)

Lock groups, 340

Lock groups and Subclasses, 341

lock-groups, 341

locking, 292

logging, 161

    Apache Commons Logging, 164

    channels, 161

- default, 162
- disabling, 163
- JDK 1.4 configuration, 164
- Log4j configuration, 163

## M

- makeDirty, 33
- makePersistenceAll, 44
- Managed Inverses, 188
- management, 309
  - configuring logging, 313
  - configuring remote, 311, 312
  - configuring WebLogic 8.1, 312
  - kodo.ManagementConfiguration, 311
- Management Console, 313
- mapping data, 210
- mapping factory, 210, 214
  - import/export mapping data, 215
- mapping files, 87
- mappings
  - class maps, 219
  - custom field, 273
  - embedded one-to-one, 248
  - external values, 276
  - externalization, 273
  - joins, 218
  - many-to-many, 253
  - many-to-many map, 264
  - many-to-N map, 263
  - many-to-pc map, 272
  - n-to-many map, 261
  - N-to-PC map, 267
  - non-primary key joins, 219
  - nonstandard joins, 218
  - one-to-many, 256
  - one-to-one, 242
  - one-to-one mapping, 251
  - partial primary key joins, 219
  - PC map, 266
  - PC one-to-one mapping, 246
  - PC-to-Many map, 270
  - PC-to-N map, 269
- mappingtool, 86, 87, 209, 215
  - ant task, 349
  - buildSchema, 211
- MBean, 309, 316
  - Datastore Cache, 318
  - PreparedStatement, 318
  - Query Cache, 318
  - TimeWatch, 318
- Mbean, 318
  - Kodo Pooling Datasource, 318
  - Log, 318
  - Runtime, 319
- metadata extensions
  - data-cache, 331
  - data-cache-timeout, 330
  - detachable, 297, 303
  - detached-objectid-field, 298

- detached-state-field, 298
- Object-Relational Mapping, 203
  - relation extensions, 198
- Metadata Tool, 197
  - ant task, 349
- metadata value indicator, 233
- monitoring (see management)
- Mutable Field Types (see Field Types)
- mutable second class object fields, 189

## N

- Named Queries, 72
  - DTD, 72
  - examples, 73
  - SQL, 79
- newQuery, 49
- No-Arg Constructor, 17
- null-value attribute, 30
- Numeric Identity, 21

## O

- object filtering
  - (see also Query)
  - advanced
    - (see also Query)
- Object Identity
  - numeric, 21
  - qualitative, 21
- Object-Relational Mapping, 209
  - buildSchema, 211
  - mapping tool, 209
  - meet-in-the-middle, 211
  - metadata extensions, 203
  - object-to-schema, 210
  - schema-to-object, 211
- object-relational mapping information, 87
- offline, 297
- one-table-per-leaf inheritance (see horizontal inheritance mapping)

## P

- Patterns
  - Data Transfer Object, 297
  - Value Object, 297
- PC collection mapping, 259
- Persistence Mechanisms, 12
- Persistence-Aware, 17
- persistence-capable (see PersistenceCapable)
- persistence-modifier attribute, 29
- PersistenceCapable, 16, 17
  - getObjectById, 47
  - Lifecycle Management, 44
  - newQuery, 49
  - Query Factory, 48
  - toString, 47
- PersistenceManager, 43, 89
  - close, 49
  - connection configuration, 38
  - properties, 37

- Uniqueness Requirement, 22
- PersistenceManagerFactory, 36, 36, 37, 41
- Persistent Class Inheritance, 17
- Persistent Class Restrictions, 17
- Persistent Classes, 184, 196
- Persistent data, 11
- Persistent Fields, 18
- persisting objects, 89
- plugin configuration, 140
- primary-key attribute, 30
- Projections (see Query)

## Q

- Qualitative Identity, 21
- Query
  - (see also JDOQL)
  - aggregates, 66
    - extensions, 292
  - by example, 60
  - candidate class, 55
  - candidate objects, 55
  - close, 62
  - closeAll, 62
  - compile, 62
  - declareImports, 58
  - declareParameters, 58
  - declareVariables, 58
  - distinct, 65
  - element, key, and value results, 65
  - execute, 62
  - executeWithArray, 62
  - executeWithMap, 62
  - extensions, 290
    - aggregates, 292
    - JDOQL, 291
    - subqueries, 323
  - filter string, 55
  - grouping, 67
  - having, 67
  - implicit parameters, 58
  - implicit variables, 58
  - imports, 58
  - JDOQL, 56
    - extensions, 291
  - limits, 63
  - named
    - (see also Named Queries)
  - object filtering, 54
    - advanced, 58
  - ordering, 63
  - parameter declarations, 58
  - projections, 64
  - result class, 68
  - result range, 63
  - setCandidates, 55
  - setClass, 55
  - setFilter, 55
  - setGrouping, 67
  - setOrdering, 63

- setRange, 63
- setResultClass, 68
- setUnique, 62
- single-string, 70
- SQL, 76
  - creating SQL queries, 76
  - named
    - (see also Named Queries)
  - projections, 78
  - retriving persistent objects, 77
  - stored procedures, 77
- subqueries, 323
- unique, 62
- variable declarations, 58
- variable results, 65
- Query Factory, 48

## R

- remote, 297, 303
- Remote Event Notification Framework, 335
- RemoteCommitListener, 336
- RemoteCommitProvider, 336
- Reverse Mapping, 98, 193

## S

- schema, 278
- schema generator, 279
- schema tool, 279, 280
  - ant task, 350
- schema-to-XML conversion (see schemagen)
- SchemaFactory, 278
- schemagen, 99
  - ant task, 350
- sequence factory, 185
- sequence-assigned, 188
- Single Field Identity, 26, 185
- single-table inheritance (see flat inheritance mapping)
- SQL
  - Obtaining a DDL for review, 282
  - queries, 76
    - creating, 76
    - for column values, 78
    - for persistent objects, 77
- SQL Batching, 326
- SQL joins, 177
- state image indicator, 231
- Stored Procedures
  - queries, 77
- subclass-join indicator, 235

## T

- TCP provider, 336
- Transaction, 50, 89
  - ACID properties, 50
  - atomicity, 50
  - begin, 51
  - commit, 51
  - consistency, 50

durability, 50

isActive, 52

isolation, 50

rollback, 51

Transparent persistence, 11

## U

Uniqueness Requirement, 22

User-defined Field Types (see Field Types)

## V

value mapping, 236

version date indicator, 230

version indicator, 229, 232

version number indicator, 229

vertical inheritance mapping, 222, 223, 224