

SolarMetric Kodo™ JDO 3.0.3 Developers Guide

Copyright © 2001, 2002, 2003, 2004 SolarMetric Inc.

SolarMetric Kodo™ JDO 3.0.3 Developers Guide

Copyright © 2001, 2002, 2003, 2004 SolarMetric Inc.

Table of Contents

I. Introduction	1
1. SolarMetric Kodo JDO	3
1.1. About This Document	4
2. SolarMetric Kodo JDO Installation	5
2.1. Overview	6
2.2. Updates	7
2.3. Key Files in the download	8
2.4. Quick Start	9
2.5. Upgrading from Kodo 2	10
2.6. Requirements	11
2.7. Terminology	12
2.8. Windows Installation (installer)	13
2.9. Windows Installation (no installer -- zip file)	14
2.10. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation	15
2.11. Borland JBuilder	16
2.12. Installing Kodo into JBuilder	17
2.13. Common installation problems	18
2.14. Resources	19
2.15. Sales Inquiries	20
II. Java Data Objects	21
1. Introduction	24
1.1. Intended Audience	25
1.2. Transparent Persistence	26
2. Why JDO?	27
3. JDO Architecture	29
3.1. JDO Exceptions	31
4. PersistenceCapable	32
4.1. JDO Enhancer	33
4.2. Persistence-Capable vs. Persistence-Aware	34
4.3. Restrictions on Persistent Classes	35
4.3.1. Default No-Arg Constructor	35
4.3.2. Inheritance	35
4.3.3. Persistent Fields	35
4.3.4. Conclusions	37
4.4. InstanceCallbacks	38
4.5. JDO Identity	40
4.5.1. Datastore Identity	40
4.5.2. Application Identity	41
4.6. Conclusions	44
5. Metadata	45
5.1. Metadata DTD	46
5.2. Metadata Placement	51
6. JDOHelper	52
6.1. Persistence-Capable Operations	53
6.2. Lifecycle Operations	54
6.3. PersistenceManagerFactory Construction	58
7. PersistenceManagerFactory	59
7.1. Obtaining a PersistenceManagerFactory	60
7.2. PersistenceManagerFactory Properties	61
7.2.1. Connection Configuration	61
7.2.2. PersistenceManager and Transaction Defaults	62
7.3. Obtaining PersistenceManagers	65
7.4. Properties and Supported Options	66

8. PersistenceManager	68
8.1. User Object Association	70
8.2. Configuration Properties	71
8.3. Transaction Association	72
8.4. Persistence-Capable Lifecycle Management	73
8.5. Lifecycle Examples	75
8.6. JDO Identity Management	77
8.7. Extent Factory	78
8.8. Query Factory	79
8.9. Closing	80
9. Transaction	81
9.1. Transaction Types	82
9.2. The JDO Transaction Interface	83
10. Extent	85
11. Query	87
11.1. Required Query Elements	88
11.2. Optional Query Elements	89
11.3. JDOQL	90
11.4. Executing Queries	94
11.5. Query Compilation	95
12. Conclusion	96
III. Kodo JDO Tutorials	97
1. Kodo JDO Tutorials	99
1.1. Tutorial Requirements	100
2. Kodo JDO Tutorial	101
2.1. The Pet Shop	102
2.1.1. Included Files	102
2.1.2. Important Utilities	102
2.2. Getting Started	104
2.2.1. Configuring the Data Store	104
2.3. Inventory Maintenance	106
2.3.1. Persisting Objects	107
2.3.2. Deleting Objects	108
2.4. Inventory Growth	110
2.5. Behavioral Analysis	112
2.5.1. Complex Queries	115
2.6. Extra Features	117
3. Reverse Mapping Tool Tutorial	118
3.1. Magazine Shop	119
3.2. Setup	120
3.2.1. Tutorial Files	120
3.2.2. Important Utilities	120
3.3. Generating Persistent Classes	121
3.4. Using the Finder	123
4. J2EE Tutorial	125
4.1. Prerequisites for the Kodo J2EE Tutorial	126
4.2. J2EE Installation Types	127
4.3. Installing Kodo JCA	128
4.3.1. JBoss 3.0	128
4.3.2. JBoss 3.2	128
4.3.3. WebLogic 6.2 to 7.x	128
4.3.4. WebLogic 8.1	129
4.3.5. WebSphere 5	129
4.3.6. SunONE Application Server	130
4.3.7. Macromedia JRun 4	130
4.3.8. Borland Enterprise Server 5.2	131
4.4. Installing the J2EE Sample Application	133
4.4.1. Compiling and Building The Sample Application	133
4.4.2. Deploying Sample To JBoss	133

4.4.3. Deploying Sample To WebLogic 6.2 to 7.x	134
4.4.4. Deploying Sample To WebLogic 8.1	134
4.4.5. Deploying Sample To SunONE	134
4.4.6. Deploying Sample To JRun	134
4.4.7. Deploying Sample To WebSphere	134
4.4.8. Deploying Sample To Borland Enterprise Server 5.2	135
4.5. Using The Sample Application	136
4.6. Sample Architecture	137
4.7. Code Notes and J2EE Tips	138
IV. Kodo JDO Frequently Asked Questions	140
1. Kodo JDO Frequently Asked Questions	142
1.1. General	143
1.2. Database	145
1.3. Programming with Kodo	147
1.4. How do I ... ?	150
1.5. Common errors	152
1.6. Productivity tools	153
1.7. Performance	154
1.8. Scalability	156
1.9. Application servers	157
1.10. Locking	158
1.11. Transactions	159
V. Kodo JDO Reference Guide	160
1. Introduction	167
1.1. Intended Audience	168
2. Configuration	169
2.1. Introduction	170
2.2. Runtime Configuration	171
2.3. Command Line Configuration	172
2.3.1. Code Formatting	172
2.4. Plugin Configuration	173
2.5. JDO Standard Properties	175
2.5.1. javax.jdo.PersistenceManagerFactoryClass	175
2.5.2. javax.jdo.option.ConnectionDriverName	175
2.5.3. javax.jdo.option.ConnectionFactoryName	175
2.5.4. javax.jdo.option.ConnectionFactory2Name	175
2.5.5. javax.jdo.option.ConnectionPassword	176
2.5.6. javax.jdo.option.ConnectionURL	176
2.5.7. javax.jdo.option.ConnectionUserName	176
2.5.8. javax.jdo.option.IgnoreCache	176
2.5.9. javax.jdo.option.Multithreaded	176
2.5.10. javax.jdo.option.NontransactionalRead	177
2.5.11. javax.jdo.option.NontransactionalWrite	177
2.5.12. javax.jdo.option.Optimistic	177
2.5.13. javax.jdo.option.RestoreValues	177
2.5.14. javax.jdo.option.RetainValues	178
2.6. Kodo JDO Properties	179
2.6.1. kodo.AggregateListeners	179
2.6.2. kodo.ClassResolver	179
2.6.3. kodo.ConnectionProperties	179
2.6.4. kodo.ConnectionFactoryProperties	179
2.6.5. kodo.Connection2DriverName	180
2.6.6. kodo.Connection2Password	180
2.6.7. kodo.Connection2URL	180
2.6.8. kodo.Connection2UserName	180
2.6.9. kodo.Connection2Properties	181
2.6.10. kodo.ConnectionFactory2Properties	181
2.6.11. kodo.ConnectionRetainMode	181
2.6.12. kodo.CopyObjectIds	181

2.6.13. kodo.DataCache	182
2.6.14. kodo.DataCacheTimeout	182
2.6.15. kodo.EagerFetchMode	182
2.6.16. kodo.FetchBatchSize	182
2.6.17. kodo.FetchGroups	182
2.6.18. kodo.FilterListeners	183
2.6.19. kodo.FlushBeforeQueries	183
2.6.20. kodo.LicenseKey	183
2.6.21. kodo.ManagedRuntime	183
2.6.22. kodo.ManagementServer	184
2.6.23. kodo.ManagementUI	184
2.6.24. kodo.PersistenceManagerImpl	184
2.6.25. kodo.PersistentClasses	184
2.6.26. kodo.ProxyManager	185
2.6.27. kodo.QueryCache	185
2.6.28. kodo.QueryCompilationCache	185
2.6.29. kodo.RemoteCommitProvider	185
2.6.30. kodo.RestoreMutableValues	186
2.6.31. kodo.RetainValuesInOptimistic	186
2.6.32. kodo.TransactionMode	186
2.6.33. kodo.jdbc.AutoIncrementConstraints	186
2.6.34. kodo.jdbc.ClassIndicator	187
2.6.35. kodo.jdbc.ConnectionDecorators	187
2.6.36. kodo.jdbc.DataSourceMode	187
2.6.37. kodo.jdbc.DBDictionary	187
2.6.38. kodo.jdbc.FetchDirection	188
2.6.39. kodo.jdbc.ForeignKeyConstraints	188
2.6.40. kodo.jdbc.JDBCListeners	188
2.6.41. kodo.jdbc.LRSSize	188
2.6.42. kodo.jdbc.MappingFactory	189
2.6.43. kodo.jdbc.ResultSetType	189
2.6.44. kodo.jdbc.SchemaFactory	189
2.6.45. kodo.jdbc.Schemas	189
2.6.46. kodo.jdbc.SequenceFactory	190
2.6.47. kodo.jdbc.SubclassMapping	190
2.6.48. kodo.jdbc.TransactionIsolation	190
2.6.49. kodo.jdbc.UpdateManager	190
2.6.50. kodo.jdbc.VersionIndicator	191
3. Logging	192
3.1. Logging Channels	193
3.2. Disabling Logging	194
3.3. Log4J	195
3.4. JDK 1.4 java.util.logging	196
3.5. Simple Log	197
3.6. Custom Log	198
4. JDBC	199
4.1. Using the Kodo JDO DataSource	200
4.2. Using a Third-Party DataSource	202
4.2.1. Enlisted Data Sources	202
4.3. Database Support	203
4.3.1. MySQLDictionary parameters	207
4.3.2. OracleDictionary parameters	207
4.4. Configuring the DBDictionary	208
4.5. Accessing Multiple Databases	209
4.6. Setting the Transaction Isolation	210
4.7. Setting the SQL Join Syntax	211
4.8. Configuring the Use of JDBC Connections	212
4.9. Runtime Access to JDBC Connections	214
4.10. Large Result Sets	215

4.11. SQL Statement Ordering & Foreign Keys	218
5. Persistent Classes	219
5.1. Restrictions on Persistent Classes	220
5.2. Object Identity	221
5.2.1. Datastore Identity	221
5.2.2. Application Identity	221
5.2.3. Primary Key Generation	222
5.2.3.1. Sequence Factory	222
5.2.3.2. Auto-Increment	223
5.3. Mutable Second Class Object Fields	226
5.3.1. Restoring Mutable Fields	226
5.3.2. Typing and Ordering	226
5.3.3. Proxies	226
5.3.3.1. Smart Proxies	226
5.3.3.2. Large Result Set Proxies	227
5.3.3.3. Custom Proxies	228
5.4. Enhancement	229
5.5. Auto-Generating Classes from a Schema	230
5.5.1. Customizing Reverse Mapping	232
5.6. Persistent Class List	234
6. Metadata	235
6.1. Generating Default JDO Metadata	236
6.2. JDO Metadata Extensions	237
6.2.1. Relation Extensions	237
6.2.1.1. inverse-owner	237
6.2.1.2. dependent	238
6.2.1.3. element-dependent	238
6.2.1.4. value-dependent	238
6.2.1.5. key-dependent	238
6.2.1.6. type	238
6.2.1.7. element-type	238
6.2.1.8. value-type	239
6.2.1.9. key-type	239
6.2.1.10. lrs	239
6.2.1.11. Example	239
6.2.2. Schema Extensions	239
6.2.2.1. jdbc-size	239
6.2.2.2. jdbc-element-size	239
6.2.2.3. jdbc-value-size	240
6.2.2.4. jdbc-key-size	240
6.2.2.5. jdbc-indexed	240
6.2.2.6. jdbc-element-indexed	240
6.2.2.7. jdbc-value-indexed	240
6.2.2.8. jdbc-key-indexed	240
6.2.2.9. jdbc-ref-indexed	240
6.2.2.10. jdbc-version-ind-indexed	240
6.2.2.11. jdbc-class-ind-indexed	240
6.2.2.12. jdbc-delete-action	241
6.2.2.13. jdbc-element-delete-action	241
6.2.2.14. jdbc-value-delete-action	241
6.2.2.15. jdbc-key-delete-action	241
6.2.2.16. jdbc-ref-delete-action	241
6.2.2.17. Example	242
6.2.3. Object-Relational Mapping Extensions	242
6.2.3.1. jdbc-class-map-name	242
6.2.3.2. jdbc-version-ind-name	242
6.2.3.3. jdbc-class-ind-name	243
6.2.3.4. jdbc-field-map-name	243
6.2.3.5. jdbc-ordered	243

6.2.3.6. jdbc-container-meta	243
6.2.3.7. jdbc-null-ind	243
6.2.3.8. externalizer	243
6.2.3.9. factory	243
6.2.3.10. jdbc-class-ind-value	244
6.2.3.11. Example	244
6.2.4. Miscellaneous Extensions	244
6.2.4.1. fetch-group	244
6.2.4.2. data-cache	244
6.2.4.3. data-cache-timeout	244
6.2.4.4. jdbc-sequence-factory	245
6.2.4.5. jdbc-sequence-name	245
6.2.4.6. jdbc-auto-increment	245
6.2.4.7. Example	245
7. Object-Relational Mapping	246
7.1. Mapping Tool	247
7.1.1. Using the Mapping Tool	248
7.2. Mapping Factory	251
7.2.1. Importing and Exporting Mapping Data	252
7.3. Mapping File XML Format	253
7.4. Mapping Notes	255
7.4.1. Join Attributes	255
7.4.2. Non-Standard Joins	256
7.5. Class Mapping	258
7.5.1. Base Mapping	258
7.5.2. Flat Mapping	259
7.5.3. Vertical Mapping	259
7.5.4. Custom Class Mapping	260
7.6. Version Indicator	262
7.6.1. Version Number Indicator	262
7.6.2. Version Date Indicator	263
7.6.3. State Image Indicator	264
7.6.4. Custom Version Indicator	264
7.7. Class Indicator	266
7.7.1. In-Class-Name Indicator	266
7.7.2. Metadata Value Indicator	267
7.7.3. Custom Class Indicator	268
7.8. Field Mapping	269
7.8.1. Value Mapping	269
7.8.2. Blob Mapping	271
7.8.3. Clob Mapping	272
7.8.4. Byte Array Mapping	274
7.8.5. One-to-One Mapping	275
7.8.6. PC One-to-One Mapping	279
7.8.7. Embedded One-to-One Mapping	280
7.8.8. Collection Mapping	282
7.8.9. Many-to-Many Mapping	284
7.8.10. One-to-Many Mapping	287
7.8.11. PC Collection Mapping	289
7.8.12. Map Mapping	291
7.8.13. N-to-Many Map Mapping	292
7.8.14. Many-to-N Map Mapping	294
7.8.15. Many-to-Many Map Mapping	295
7.8.16. PC Map Mapping	297
7.8.17. N-to-PC Map Mapping	298
7.8.18. PC-to-N Map Mapping	300
7.8.19. PC-to-Many Map Mapping	301
7.8.20. Many-to-PC Map Mapping	302
7.8.21. Custom Field Mapping	304

7.8.22. Externalization	304
8. Schema Information	308
8.1. Schema Reflection	309
8.1.1. Schemas List	309
8.1.2. Schema Factory	309
8.1.3. Schema Generator	310
8.2. Schema Tool	312
8.3. XML Schema Format	314
8.4. The SQLLine Utility	316
9. Runtime Deployment	318
9.1. JDOHelper	319
9.2. KodoHelper	320
9.3. J2EE Deployment	321
9.4. Using Kodo JDO via the Java Connector Architecture	323
9.4.1. Deploying on JBoss 3.0	323
10. JDO Runtime Extensions	324
10.1. KodoPersistenceManagerFactory	325
10.2. KodoPersistenceManager	326
10.2.1. JDO Events	326
10.2.2. PersistenceManager Extension	326
10.3. KodoExtent	327
10.4. KodoQuery	328
10.5. Fetch Configuration	329
10.6. KodoHelper	330
10.7. Query Extensions	331
10.7.1. JDOQL Extensions	331
10.7.1.1. Included Query Extensions	331
10.7.1.2. Developing Custom Query Extensions	333
10.7.1.3. Configuring JDOQL Extensions	333
10.8. Query Aggregates and Projections	334
10.8.1. Query Aggregates	334
10.8.1.1. Using Query Aggregates	334
10.8.1.2. Included Query Aggregates	335
10.8.1.2.1. count	335
10.8.1.2.2. sum	335
10.8.1.2.3. min	335
10.8.1.2.4. max	336
10.8.1.2.5. avg	336
10.8.1.3. Developing Custom Query Aggregates	336
10.8.1.4. Configuring Query Aggregates	336
10.8.2. Query Projections	336
10.8.2.1. Using Query Projections	336
10.8.3. JDOQL Variables in Aggregates and Projections	337
10.8.4. Custom Query Result Class	338
10.8.4.1. Using a Bean Query Result Class	338
10.8.4.2. Using a Generic Query Result Class	338
10.8.4.3. Using a Result Alias	339
11. Detach and Attach	340
11.1. Configuring detachability	341
11.2. Detachable behavior	342
11.3. Detach and Attach Callbacks	344
12. Management and Monitoring	345
12.1. Configuring Local Management / Monitoring	346
12.2. Configuring Remote Management / Monitoring	347
12.3. Configuring Logging for Management / Monitoring	348
13. Enterprise Edition	349
13.1. Integrating with the Transaction Manager	350
13.2. XA Transactions	351
13.2.1. Requirements for Using Kodo with XA Transactions	351

13.2.2. Configuring Kodo to Utilize XA Transactions	351
13.3. Direct SQL Execution	353
13.4. MethodQL	354
14. Performance Pack	355
14.1. SQL Batching	356
14.2. Eager Fetching	357
14.2.1. Configuring Eager Fetching	357
14.2.2. Eager Fetching Considerations	358
14.3. Datastore Cache	359
14.3.1. Overview of Kodo JDO Datastore Caching	359
14.3.2. Kodo JDO Cache Usage	359
14.3.3. Query Caching	362
14.3.4. Tangosol Integration	363
14.3.5. Cache Extension	364
14.3.6. Important Notes	364
14.3.7. Known Issues and Limitations	364
14.4. Remote Event Notification Framework	366
14.4.1. Remote Commit Provider Configuration	366
14.4.2. Customization	367
14.5. Fetch Groups	368
14.5.1. Normal Default Fetch Group Behavior	368
14.5.2. Kodo JDO Fetch Group Behavior	369
14.5.3. Custom Fetch Group Configuration	369
15. Third Party Integration	371
15.1. Overview of Third Party Integration features in Kodo	372
15.2. Apache Ant	373
15.2.1. Common Ant Configuration Options	373
15.2.2. JDO Enhancer Ant Task	375
15.2.3. Application Identity Tool Ant Task	375
15.2.4. JDO Metadata Tool Ant Task	375
15.2.5. Mapping Tool Ant Task	376
15.2.6. Reverse Mapping Tool Ant Task	376
15.2.7. Schema Tool Ant Task	377
15.2.8. Schema Generator Ant Task	377
15.3. XDoclet	378
15.4. Borland JBuilder	383
15.4.1. Installing Kodo Into JBuilder	383
15.4.2. Kodo Configuration from JBuilder	383
15.4.3. Creating and building JDO projects in JBuilder	383
15.4.4. Editing JDO Metadata from JBuilder	384
15.4.5. Editing Mapping Info from JBuilder	384
15.4.6. JBuilder Project Sample	384
15.5. Sun ONE Studio / NetBeans IDE	386
15.5.1. Before Installing Kodo into the IDE	386
15.5.2. Installing Kodo into the IDE	386
15.5.3. Configuring the Kodo Module	386
15.5.4. Kodo Template Wizards	387
15.5.5. JDO DataObject	387
15.5.6. Mapping DataObject	387
15.5.7. Kodo Integration into the Build Process	387
15.5.8. SunONE / NetBeans Sample	387
15.6. Eclipse / WebSphere Studio Integration	389
15.6.1. Installing the Kodo Eclipse Plugin	389
15.6.2. Configuring the Plugin	389
15.6.3. Using Kodo in Eclipse IDEs	390
15.6.4. Eclipse Sample	390
16. Optimization Techniques	391
VI. Kodo JDO Examples	396
1. Kodo Sample Code	398

1.1. Using Application Identity	399
1.2. Using JDO with Java Server Pages (jsp)	400
1.3. Custom Proxies	401
1.4. JDO Enterprise Java Beans Facade	402
1.5. Customizing Logging	403
1.6. Custom Sequence Factory	404
1.7. Using Externalization to Persist Second Class Objects	405
1.8. Using Persistent Classes Without Enhancement	406
1.9. XDoclet Integration	407
1.10. Custom Mappings	408
1.11. Example of full-text searching in JDO	409
1.12. Management / Monitoring	410
VII. Kodo Development Workbench Guide	412
1. Introduction to the Kodo Development Workbench	cdxiv
1.1. Kodo Development Workbench Requirements	cdxv
1. Running and Configuring Kodo Development Workbench	416
1.1. Starting Kodo Development Workbench From the Command Line	417
1.2. Configuring Kodo Development Workbench	418
1.3. Standalone Configuration Tool	419
2. Getting Started with Kodo Development Workbench	420
2.1. Beginning The Kodo Development Workbench Tutorial	421
2.2. Getting Familiar with Kodo Development Workbench	422
2.3. The MetaData Explorer	423
2.4. The Schema Explorer	424
2.5. Kodo Development Workbench Logging	425
2.6. The Editor	426
2.7. Running The Tutorial	427
3. Root MetaData Actions	428
3.1. Mount JDO File	429
3.2. Unmount Files...	430
3.3. Create MetaData	431
3.4. Import Mapping Info	432
4. MetaData Actions	433
4.1. Enhance	434
4.2. Edit MetaData	435
4.3. Add - Recreate Mapping Info	436
4.4. Export Mapping Info	437
4.5. Drop Mapping Info	438
4.6. Remove MetaData	439
4.7. Build Schema From Mapping	440
4.8. Visualize Mapping	441
5. Root Schema Actions	442
5.1. Run SchemaTool	443
5.2. Refresh Schema From DB	444
5.3. Create DB Script	445
5.4. Create Change Script	446
5.5. Reverse Map Schema	447
5.6. Import .schema file	448
5.7. Export to .schema file	449
6. Schema Actions	450
6.1. Drop Schema Object	451
6.2. Edit Table	452
6.3. Add	453
7. The Editors	454
7.1. MetaData Editor	455
7.2. The Mapping Editor	456
7.3. The Table Editor	457
A. JDO Resources	458
B. Supported Databases	459

B.1. JDataStore	460
B.2. IBM DB2	461
B.2.1. Known issues with DB2	461
B.3. Hypersonic	462
B.3.1. Known issues with Hypersonic	462
B.4. Informix	463
B.4.1. Known issues with Informix	463
B.5. Microsoft Access	464
B.5.1. Known issues with Microsoft Access	464
B.6. Microsoft SQL Server	465
B.6.1. Known issues with SQL Server	465
B.7. Microsoft FoxPro	466
B.7.1. Known issues with Microsoft FoxPro	466
B.8. MySQL	467
B.8.1. Known issues with MySQL	467
B.9. Oracle	468
B.9.1. Known issues with Oracle	468
B.10. Pointbase	469
B.10.1. Known issues with Pointbase	469
B.11. PostgreSQL	470
B.11.1. Known issues with PostgreSQL	470
B.12. Sybase Adaptive Server	471
B.12.1. Known issues with Sybase	471
C. Common Database Errors	472
D. Migrating from Kodo 2 to Kodo 3	481
D.1. Source Code Migration	482
D.1.1. Package Structure Changes	482
D.1.2. API Changes	482
D.2. JDO Metadata Migration	484
D.3. Properties File Migration	486
D.4. Storing Object-Relational Mapping Data	487
D.5. Kodo 3 Development Process	488
E. Implementation Notes	489
E.1. jdoFlags Fields in the Default Fetch Group	490
F. Development and Runtime Libraries	491
G. Release Notes	493
H. Known Bugs and Limitations	521

List of Tables

- 2.1. Persistence Mechanisms 27
- 6.1. JDOHelper Lifecycle Methods 57
- 2.1. Pre-defined aliases 185
- 4.1. Validation SQL Defaults 201
- 4.2. Kodo Automatic Flush Behavior 212
- 7.1. Externalizer Options 305
- 7.2. Factory Options 305
- 14.1. Data access methods 359
- 16.1. Optimization Techniques 391
- 4.1. Graph Edges 441
- B.1. Supported Databases and JDBC Drivers 459
- C.1. Known Database Error Codes 472
- D.1. Notable Package Changes 482

List of Examples

3.1. Interaction of JDO Interfaces	30
4.1. PersistenceCapable Class	32
4.2. Accessing Mutable Persistent Fields	37
4.3. Using the InstanceCallbacks Interface	38
4.4. JDO Identity Objects	40
4.5. Application Identity Class	42
5.1. Basic Structure of Metadata Documents	46
5.2. Metadata Class Listings	47
5.3. Complete Metadata Document	49
6.1. Obtaining a PersistenceManagerFactory	58
8.1. Persisting Objects	75
8.2. Updating Objects	75
8.3. Deleting Objects	75
9.1. Grouping Operations with Transactions	84
10.1. Iterating an Extent	85
11.1. Basic Query	91
11.2. Result Ordering and Method Calls	91
11.3. Mathematical Operations and Relation Traversal	91
11.4. Precedence and Logical Operators	91
11.5. Imports and Parameters	92
11.6. Collections	92
11.7. Variables	92
11.8. Collection Parameters	93
11.9. Processing a Query Result	94
1.1. Issuing a query against a Date field	150
2.1. Code Formatting with the Application Id Tool	172
3.1. Standard Logging	195
3.2. Quiet Logging	195
3.3. Verbose Logging	195
3.4. Log Properties	196
3.5. Example Simple Log Properties	197
3.6. Custom Logging Class	198
4.1. Properties File for the Kodo JDO DataSource	201
4.2. Properties File for a Third-Party DataSource	202
4.3. Specifying a DBDictionary	208
4.4. Specifying a Transaction Isolation	210
4.5. Specifying the Join Syntax Default	211
4.6. Specifying the Join Syntax at Runtime	211
4.7. Specifying Connection Usage Defaults	213
4.8. Specifying Connection Usage at Runtime	213
4.9. Obtaining a JDBC Connection from the PersistenceManager	214
4.10. Obtaining a JDBC Connection from the DataSource	214
4.11. Specifying Result Set Defaults	216
4.12. Specifying Result Set Behavior at Runtime	216
4.13. Using Random Access Query Results in a Portable Fashion	216
4.14. Enabling SQL Statement Ordering	218
5.1. Using the Application Identity Tool	221
5.2. Sequence Factory Configuration	223
5.3. Accessing the Sequence Factory	223
5.4. Auto-Increment Metadata	224
5.5. Auto-Increment Object Ids	224
5.6. Using Initial Field Values	226
5.7. Using a Large Result Set Iterator	227
5.8. Marking a Large Result Set Field	228

5.9. Configuring the Proxy Manager	228
5.10. Using the Kodo JDO Enhancer	229
5.11. Using the Schema Generator	230
5.12. Using the Reverse Mapping Tool	230
5.13. Using the Mapping Tool	232
5.14. Customizing Reverse Mapping with Properties	232
6.1. Using the MetaDataTool	236
7.1. Using the Mapping Tool	247
7.2. Refreshing Mappings and the Relational Schema	248
7.3. Dropping Mappings	249
7.4. Validating Mappings	249
7.5. Updating the Schema Based on Mapping Data	249
7.6. Modifying Default Mappings	250
7.7. Reverting Mapping Data	250
7.8. Modifying Difficult-to-Access Mapping Data	252
7.9. Switching Mapping Factories	252
7.10. Basic Structure of Mapping Documents	253
7.11. Complete Mapping Document	254
7.12. Using a Base Mapping	258
7.13. Using a Flat Mapping	259
7.14. Using a Vertical Mapping	260
7.15. Using a Version Number Indicator	262
7.16. Using a Version Date Indicator	263
7.17. Using a State Image Indicator	264
7.18. Using a In-Class-Name Indicator	266
7.19. Using a Metadata-Value Indicator	267
7.20. Using a Value Mapping	269
7.21. Using a Value Mapping in a Separate Table	270
7.22. Using a Blob Mapping	272
7.23. Using a Clob Mapping	273
7.24. Using a Byte array Mapping	274
7.25. Using a One-to-One Mapping	276
7.26. Using a Two-Sided One-to-One Mapping	276
7.27. Using a One-to-One Mapping With Inverse Columns	278
7.28. Using a PC One-to-One Mapping	279
7.29. Using an Embedded One-to-One Mapping	281
7.30. Using a Collection Mapping	283
7.31. Using a Many-to-Many Mapping	284
7.32. Using a Two-Sided Many-to-Many Mapping	285
7.33. Using a One-to-Many Mapping	287
7.34. Using a One-Sided One-to-Many Mapping	288
7.35. Using a PC Collection Mapping	290
7.36. Using a Map Mapping	291
7.37. Using an N-to-Many Map Mapping	293
7.38. Using a Many-to-N Map Mapping	294
7.39. Using a Many-to-Many Map Mapping	296
7.40. Using a PC Map Mapping	297
7.41. Using an N-to-PC Map Mapping	299
7.42. Using a PC-to-N Map Mapping	300
7.43. Using a PC-to-Many Map Mapping	301
7.44. Using a Many-to-PC Map Mapping	303
7.45. Using Externalization	306
7.46. Querying Externalization Fields	307
8.1. Using the Schema Generator	310
8.2. Schema Creation	313
8.3. SQL Scripting	313
8.4. Schema Drop	313
8.5. Basic Schema	314
8.6. Full Schema	315

8.7. Connecting to the Database	316
8.8. Examining the Tutorial Schema	316
8.9. Issuing SQL Against the Database	317
9.1. Specifying the PersistenceManagerFactory	319
9.2. Using the JDOHelper	319
9.3. Using the KodoHelper	320
9.4. Binding a PersistenceManagerFactory into JNDI via a WebLogic Startup Class	321
9.5. Looking up the PersistenceManagerFactory in JNDI	321
10.1. Basic Query Extension	331
10.2. Chaining Query Extensions	331
10.3. Basic Query Aggregate	334
10.4. Multiple Query Aggregates	334
10.5. Casting Query Aggregates	335
10.6. Basic Query Projections	337
10.7. Multiple Query Projections	337
10.8. Using a Result Class Bean	338
10.9. Using a Generic Result Class	339
10.10. Using a Result Alias	339
11.1. Detaching a Single Instance	342
11.2. Using Custom Fetch Groups for Detach	343
13.1. Configuring Transaction Manager Integration	350
13.2. XA Configuration	351
14.1. Configuring SQL Batching	356
14.2. Setting the Default Eager Fetch Mode	358
14.3. Setting the Eager Fetch Mode at Runtime	358
14.4. Specifying a DataCache Timeout	360
14.5. Specifying a Non-Default DataCache	360
14.6. Configuring and Acquiring a Named DataCache	360
14.7. Pinning an Object into the DataCache	361
14.8. Unpinning an Object from the DataCache	361
14.9. Evicting an Object from the DataCache	361
14.10. Setting the Size of the Query Cache	362
14.11. Disabling the Query Cache	362
14.12. Notifying the Query Cache of Altered Classes	362
14.13. Dropping or Pinning Query Results	363
14.14. Disabling and Enabling Query Caching	363
14.15. Distributed Tangosol Cache Configuration	364
14.16. Query Replaces Extent	365
14.17. JMS Remote Commit Provider Configuration	366
14.18. TCP Remote Commit Provider Configuration	367
14.19. Custom Fetch Group Meta-Data	369
14.20. Adding a Fetch Group to a Query	370
15.1. Using the <config> Ant Tag	373
15.2. Using the Properties Attribute of the <config> Tag	373
15.3. Using the PropertiesFile Attribute of the <config> Tag	374
15.4. Using the <classpath> Ant Tag	374
15.5. Using the <codeformat> Ant Tag	374
15.6. Invoking the JDO Enhancer from Ant	375
15.7. Invoking the Application Identity Tool from Ant	375
15.8. Invoking the JDO Metadata Tool from Ant	376
15.9. Invoking the Mapping Tool from Ant	376
15.10. Invoking the Reverse Mapping Tool from Ant	376
15.11. Invoking the Schema Tool from Ant	377
15.12. Invoking the Schema Generator from Ant	377
15.13. Commenting for XDoclet	378
15.14. Invoking XDoclet with Ant	381
16.1. Explicitly Closing Resources	393
16.2. Disabling the Class Indicator	394
16.3. Appropriate use of JDOQL parameters	394

16.4. Inappropriate use of JDOQL parameters	395
B.1. Example properties for JDataStore	460
B.2. Example properties for IBM DB2	461
B.3. Example properties for Hypersonic	462
B.4. Example properties for Informix Dynamic Server	463
B.5. Example properties for Microsoft Access	464
B.6. Example properties for Microsoft SQLServer	465
B.7. Example properties for Microsoft FoxPro	466
B.8. Example properties for MySQL	467
B.9. Example properties for Oracle	468
B.10. Example properties for Pointbase	469
B.11. Example properties for PostgreSQL	470
B.12. Example properties for Sybase	471
D.1. Using the Kodo 2 Migrator	484
D.2. Invoking the Kodo 2 Migrator from Ant	484
D.3. Using the Kodo 2 Properties Tool	486
D.4. Invoking the Kodo 2 Properties Tool from Ant	486

Part I. Introduction

Table of Contents

- 1. SolarMetric Kodo JDO 3
 - 1.1. About This Document 4
- 2. SolarMetric Kodo JDO Installation 5
 - 2.1. Overview 6
 - 2.2. Updates 7
 - 2.3. Key Files in the download 8
 - 2.4. Quick Start 9
 - 2.5. Upgrading from Kodo 2 10
 - 2.6. Requirements 11
 - 2.7. Terminology 12
 - 2.8. Windows Installation (installer) 13
 - 2.9. Windows Installation (no installer -- zip file) 14
 - 2.10. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation 15
 - 2.11. Borland JBuilder 16
 - 2.12. Installing Kodo into JBuilder 17
 - 2.13. Common installation problems 18
 - 2.14. Resources 19
 - 2.15. Sales Inquiries 20

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 3.0.3. Kodo JDO is an implementation of the Java Data Objects standard for relational databases. Version 3.0.3 supports the final release of the JDO 1.0.1 specification. A copy of this specification is provided in the documentation of Kodo JDO version 3.0.3.

You just downloaded one of the following:

- `kodo-jdo-3.0.3.tar.gz` - Kodo JDO version 3.0.3 for Unix or Mac OS X.
- `kodo-jdo-3.0.3.exe` - Kodo JDO version 3.0.3 for Microsoft Windows NT and Windows 2000. Includes Windows Installer.
- `kodo-jdo-3.0.3.zip` - Kodo JDO version 3.0.3 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/.

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 `jdocmd.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	24
1.1. Intended Audience	25
1.2. Transparent Persistence	26
2. Why JDO?	27
3. JDO Architecture	29
3.1. JDO Exceptions	31
4. PersistenceCapable	32
4.1. JDO Enhancer	33
4.2. Persistence-Capable vs. Persistence-Aware	34
4.3. Restrictions on Persistent Classes	35
4.3.1. Default No-Arg Constructor	35
4.3.2. Inheritance	35
4.3.3. Persistent Fields	35
4.3.4. Conclusions	37
4.4. InstanceCallbacks	38
4.5. JDO Identity	40
4.5.1. Datastore Identity	40
4.5.2. Application Identity	41
4.6. Conclusions	44
5. Metadata	45
5.1. Metadata DTD	46
5.2. Metadata Placement	51
6. JDOHelper	52
6.1. Persistence-Capable Operations	53
6.2. Lifecycle Operations	54
6.3. PersistenceManagerFactory Construction	58
7. PersistenceManagerFactory	59
7.1. Obtaining a PersistenceManagerFactory	60
7.2. PersistenceManagerFactory Properties	61
7.2.1. Connection Configuration	61
7.2.2. PersistenceManager and Transaction Defaults	62
7.3. Obtaining PersistenceManagers	65
7.4. Properties and Supported Options	66
8. PersistenceManager	68
8.1. User Object Association	70
8.2. Configuration Properties	71
8.3. Transaction Association	72
8.4. Persistence-Capable Lifecycle Management	73
8.5. Lifecycle Examples	75
8.6. JDO Identity Management	77
8.7. Extent Factory	78
8.8. Query Factory	79
8.9. Closing	80
9. Transaction	81
9.1. Transaction Types	82
9.2. The JDO Transaction Interface	83
10. Extent	85
11. Query	87
11.1. Required Query Elements	88
11.2. Optional Query Elements	89
11.3. JDOQL	90
11.4. Executing Queries	94
11.5. Query Compilation	95

12. Conclusion	96
----------------------	----

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

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

- *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.

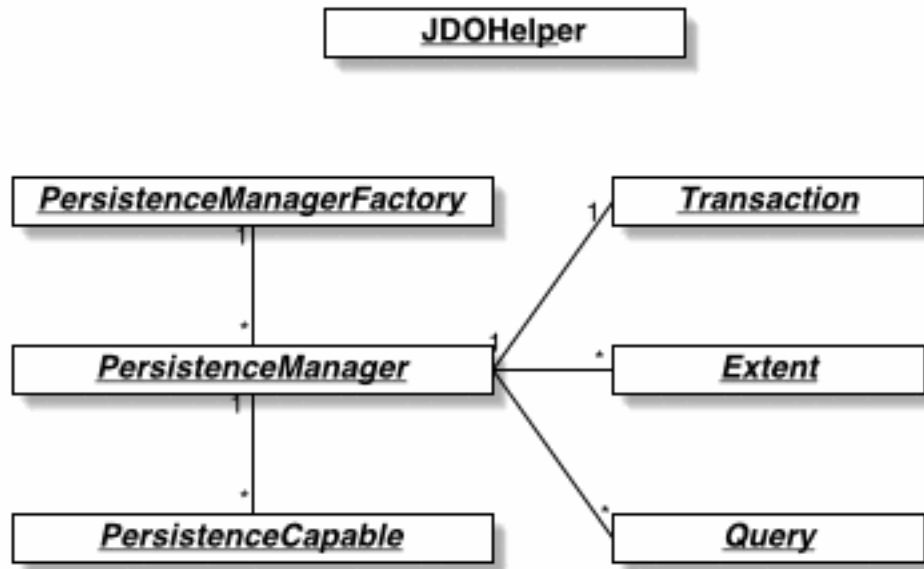
Note

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);
}

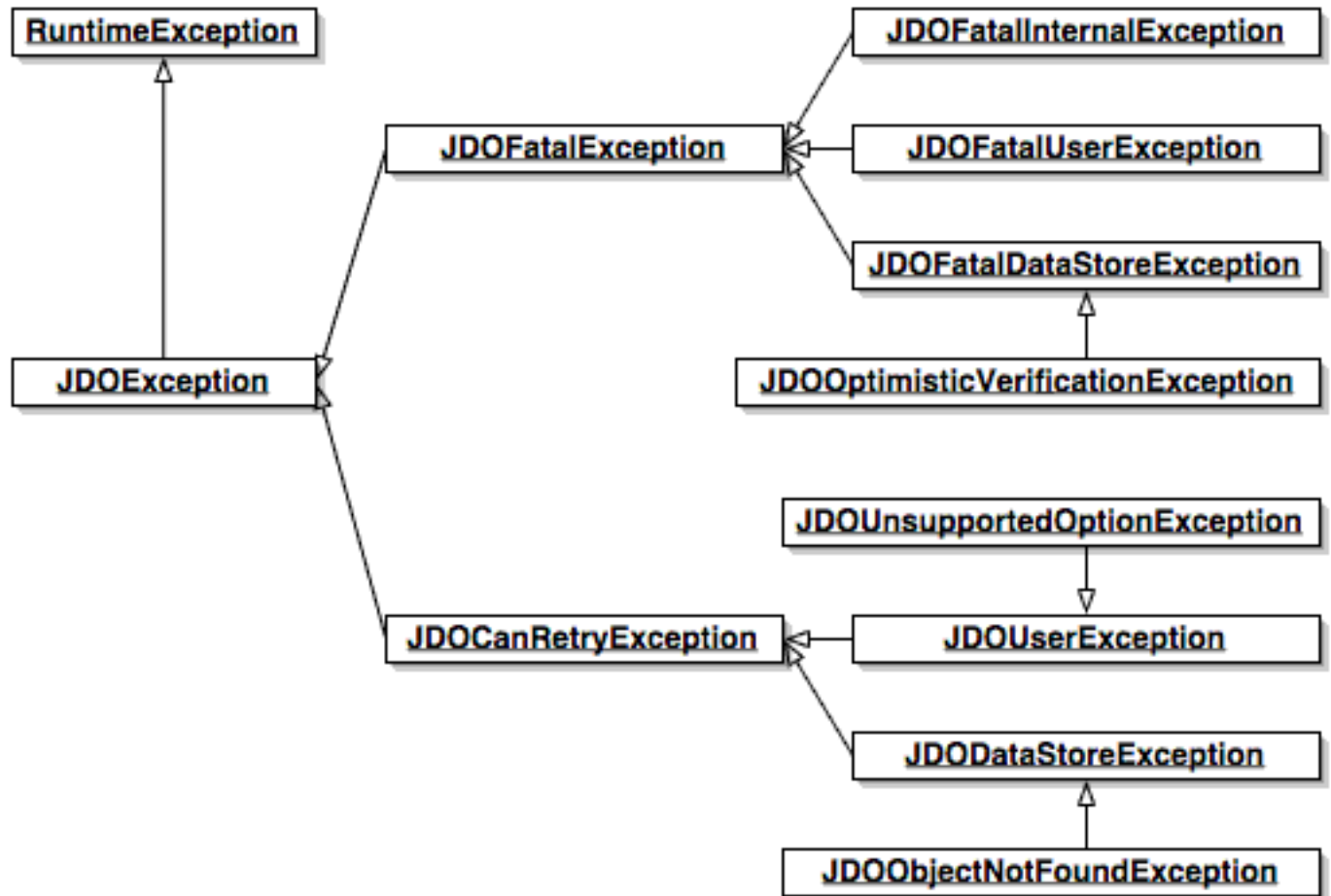
// commit the updates and free resources
tx.commit ();
pm.close ();
factory.close ();
```

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 ();

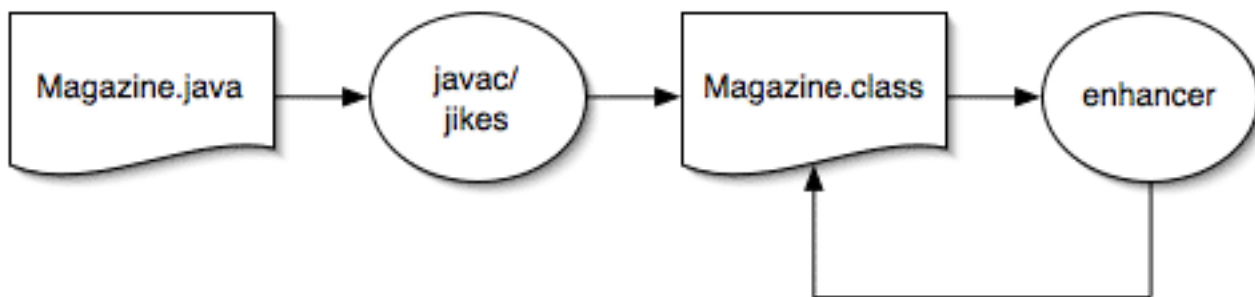
        this.publisher = publisher;
        publisher.addMagazine (this);
        this.copyright = copyright;
    }

    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 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.

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 use the same identity class. 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 `HashSet`s 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. Dirty-

ing 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.

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

- The names of the non-static fields of the class must include the names of the primary key fields of the corresponding persistence-capable class, and the field types must be identical.
- The `equals` and `hashCode` methods of the class must use the values of all fields corresponding to primary key fields in the persistence-capable class.
- If the class is an inner class, it must be `static`.
- All classes related by inheritance must use the same application identity class.
- Each inheritance tree must use a unique application identity class.
- Primary key fields must be primitives, primitive wrappers, or `Strings`. Notably, other persistent instances can *not* be used as primary key fields.

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

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

4.6. 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$inner-class`. For example, `SubscriptionForm$LineItem`.

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

- `name`: The name of the class. This attribute is required.
- `persistence-capable-superclass`: If the superclass of this class is also persistent, and you wish JDO to know about the inheritance structure, then you must name the superclass in this attribute. If the superclass of this class is not persistent or if for some reason you want JDO to treat the superclass as unrelated, you should not specify this attribute.
- `identity-type`: Gives the JDO identity type used by the class. Legal values are `application` for application identity, `datastore` for datastore identity, and `none`. This attribute defaults to a value of `application` if the `objectid-class` attribute is specified, and `datastore` otherwise.
- `objectid-class`: For application identity, the name of the JDO identity class used by this class. The `objectid-class` should only be given for the base class in the inheritance tree; subclasses will "inherit" the identity class from their superclass.
- `requires-extent`: Set this attribute to `false` if you will never need to query for persistent instances of this class (i.e., if all objects of the class can be obtained through JDO identity lookups or through relations with other objects). Defaults to `true`.

Example 5.2. Metadata Class Listings

```
<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      ...
    </class>
    <class name="Article">
      ...
    </class>
    <class name="Author">
      ...
    </class>
    <class name="Address">
      ...
    </class>
  </package>
</jdo>
```

```
</package>
<package name="org.mag.subscribe">
  <class name="Form">
    ...
  </class>
  <class name="SubscriptionForm" persistence-capable-superclass="Form">
    ...
  </class>
  <class name="SubscriptionForm$LineItem">
    ...
  </class>
</package>
</jdo>
```

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

- `name`: The name of the field, as it is declared in the persistence-capable class. This attribute is required.
- `persistence-modifier`: Specifies how JDO should manage the field. Legal values are `persistent` for persistent fields, `transactional` for fields that are non-persistent but can be rolled back along with the current **transaction**, and `none`. The default value of this attribute is based on the type of the field:
 - Fields declared `static`, `transient`, or `final` default to `none`.
 - Fields of any primitive or primitive wrapper type default to `persistent`.
 - Fields of types `java.lang.String`, `java.lang.Number`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Locale`, and `java.util.Date` default to `persistent`.
 - Fields of any user-defined persistence-capable type default to `persistent`.
 - Arrays of any of the types mentioned so far default to `persistent`.
 - Fields of the following container types in the `java.util` package default to `persistent`: `Collection`, `Set`, `List`, `Map`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, `Vector`.
 - All other fields default to `none`.
- `primary-key`: Set this attribute to `true` if the class uses application identity and this field is a primary key field. Defaults to `false`.
- `null-value`: Specifies the treatment of `null` values when the field is written to the data store. Use a value of `none` if the data store should hold a `null` value for the field. Use `default` to write a data store default value instead. Finally, use `exception` if you want the JDO implementation to throw an exception if the field contains a `null` value when it is being written to the data store. Defaults to `none`.
- `default-fetch-group`: Default fetch group fields are managed together as a group for efficiency. They are typically loaded as a block from the data store, and are often written as a block as well. This attribute defaults to `true` for primitive, primitive wrapper, `String`, `Date`, `BigDecimal`, and `BigInteger` types. All other types default to `false`.

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.

- `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`.

map elements define four attributes. They are:

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

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

Example 5.3. Complete Metadata Document

```
<?xml version="1.0"?>
<!-- Note that all persistence-capable classes must be listed, but -->
<!-- very few fields need to be specified -->
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      <field name="isbn" primary-key="true"/>
      <field name="title" primary-key="true"/>
      <field name="articles">
        <collection element-type="Article"/>
      </field>
    </class>
    <class name="Article">
      <field name="authors">
        <map key-type="String" value-type="Author"/>
      </field>
    </class>
    <class name="Author">
      <field name="address" embedded="true"/>
    </class>
  </package>
</jdo>
```

```
    </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-New-Deleted	X	X	X	X	X

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:hsql:database");
props.setProperty ("javax.jdo.option.ConnectionDriverName",
    "org.hsqldb.jdbcDriver");

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

Chapter 7. PersistenceManagerFactory

PersistenceManagerFactory

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

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

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

Collection supportedOptions()
Properties getProperties()
```

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

7.1. Obtaining a PersistenceManagerFactory

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

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

7.2. PersistenceManagerFactory Properties

The majority of the `PersistenceManagerFactory` interface consists of Java bean-style "getter" and "setter" methods for several properties, represented by field declarations in the diagram at the beginning of this chapter. These properties can be grouped into 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

PersistenceManager

Object UserObject
boolean Multithreaded
boolean IgnoreCache

Transaction currentTransaction()

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

Object getObjectById(Object oid, boolean validate)
Object getObjectId(Object pc)
Class getObjectIdClass(Class pcClass)
Object newObjectIdInstance(Class pcClass, String str)

Query newQuery(...)

Extent getExtent(Class pcClass, boolean includeSubclasses)

boolean isClosed()
void close()

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

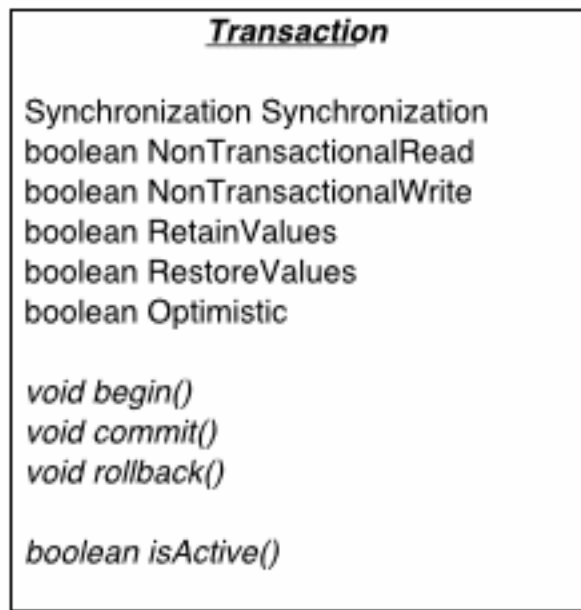
9.1. Transaction Types

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



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 ResultSets 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

Query

```
boolean ignoreCache

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

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

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

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

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

We will now explore the elements of the `Query` interface. You might find yourself scratching your head over some aspects of the discussion below; this is to be expected. We will clarify everything with several concrete examples of JDO queries later in this chapter.

11.1. Required Query Elements

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

Every Query has three required elements:

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

11.2. Optional Query Elements

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

The following are optional elements of JDO queries:

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

11.3. JDOQL

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

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

Note

Kodo JDO offers several extensions to JDOQL, and allows you to define your own extension as well. See the user guide for details.

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

```
package org.mag;

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

    ...
}

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

    ...
}

package org.mag.pub;
```

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

    ...
}
```

Example 11.1. Basic Query

Find all magazines whose price is greater than 10 dollars. Notice that we can use the candidate class' persistent fields in our filter string. JDOQL is based only on the object model, not on the underlying data store representation of the object.

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

Example 11.2. Result Ordering and Method Calls

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

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

Example 11.3. Mathematical Operations and Relation Traversal

Find all magazines whose sales account for over 1% of the total revenue for the publisher, in descending order by publisher revenue and ascending order by publisher name. Notice the use of standard "dot" syntax to traverse into the magazine's publisher relation to retrieve the revenue and name fields.

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

Example 11.4. Precedence and Logical Operators

Find all magazines published by Random House or Addison Wesley whose price is less than or equal to 10 dollars.

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

Example 11.5. Imports and Parameters

Find all magazines published by a given company whose price is greater than a given number. Notice that both the company and the number to use are represented by placeholder parameters in the filter string; their values are not supplied until the query is executed. We import the `org.mag.pub` package classes so that we can declare a parameter of type `Company`, which resides in that package. We could just as easily skipped the import and declared the parameter using the fully-qualified `org.mag.pub.Company` type. Importing or full qualification is only necessary when the declared class is not in `java.lang`, `java.util`, `java.math`, or the package of the query's candidate class.

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

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

Example 11.6. Collections

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

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

Example 11.7. Variables

Find all magazines that have an article titled "Fourier Transforms". Notice the use of a variable to represent any article in the magazine's articles collection.

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

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

Example 11.8. Collection Parameters

Find all magazines that have an article whose title is one of a number of passed-in possibilities. Here, we use a collection parameter to hold all the possible values. If you are familiar with SQL, this is JDOQL's equivalent to the SQL `IN` test.

```
Collection titles = ...;    // collection of possible titles

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

11.4. Executing Queries

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

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

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

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

Note

Query results from Kodo JDO always implement the `java.util.List` interface. This allows you to access the results in any order, and supports the common need to only process a certain range of objects in the result set. Because Kodo JDO can be configured to use scrollable JDBC `ResultSet`s where appropriate, list elements that are never accessed will not affect performance.

Combined with Kodo JDO's memory-sensitive data structures, scrollable result sets also allow 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 if you have configured Kodo to use scrolling result sets you are strongly encouraged to close your query results 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 11.9. Processing a Query Result

```
PersistenceManager pm = ...;

Query query = pm.newQuery (Employee.class, "salary > 50000");
Collection employees = (Collection) query.execute ();
try
{
    for (Iterator itr = employees.iterator (); itr.hasNext ();)
        processEmployee ((Employee) itr.next ());
}
finally
{
    query.close (employees);
}
```

11.5. Query Compilation

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

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

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

Chapter 12. 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	99
1.1. Tutorial Requirements	100
2. Kodo JDO Tutorial	101
2.1. The Pet Shop	102
2.1.1. Included Files	102
2.1.2. Important Utilities	102
2.2. Getting Started	104
2.2.1. Configuring the Data Store	104
2.3. Inventory Maintenance	106
2.3.1. Persisting Objects	107
2.3.2. Deleting Objects	108
2.4. Inventory Growth	110
2.5. Behavioral Analysis	112
2.5.1. Complex Queries	115
2.6. Extra Features	117
3. Reverse Mapping Tool Tutorial	118
3.1. Magazine Shop	119
3.2. Setup	120
3.2.1. Tutorial Files	120
3.2.2. Important Utilities	120
3.3. Generating Persistent Classes	121
3.4. Using the Finder	123
4. J2EE Tutorial	125
4.1. Prerequisites for the Kodo J2EE Tutorial	126
4.2. J2EE Installation Types	127
4.3. Installing Kodo JCA	128
4.3.1. JBoss 3.0	128
4.3.2. JBoss 3.2	128
4.3.3. WebLogic 6.2 to 7.x	128
4.3.4. WebLogic 8.1	129
4.3.5. WebSphere 5	129
4.3.6. SunONE Application Server	130
4.3.7. Macromedia JRun 4	130
4.3.8. Borland Enterprise Server 5.2	131
4.4. Installing the J2EE Sample Application	133
4.4.1. Compiling and Building The Sample Application	133
4.4.2. Deploying Sample To JBoss	133
4.4.3. Deploying Sample To WebLogic 6.2 to 7.x	134
4.4.4. Deploying Sample To WebLogic 8.1	134
4.4.5. Deploying Sample To SunONE	134
4.4.6. Deploying Sample To JRun	134
4.4.7. Deploying Sample To WebSphere	134
4.4.8. Deploying Sample To Borland Enterprise Server 5.2	135
4.5. Using The Sample Application	136
4.6. Sample Architecture	137
4.7. Code Notes and J2EE Tips	138

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* \[5\]](#) 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.

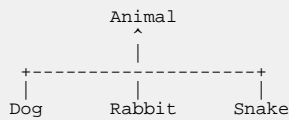
Chapter 2. Kodo JDO Tutorial

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” \[106\]](#)
- **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 \[45\]](#) 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 5.4, “Enhancement” [229]** 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* [246]** 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 \(no installer -- zip file\)” \[14\]](#) of Part I of this manual for Windows, or [Section 2.10, “POSIX \(Linux, Solaris, Mac OS X, Windows with cygwin, etc.\) Installation” \[15\]](#) of Part I of this manual for POSIX (Linux, Solaris, Cygwin, etc.). Detailed information on which libraries are needed can be found in [Appendix F, *Development and Runtime Libraries* \[491\]](#). 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.

```
jdgc 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 5.4, “Enhancement” \[229\]](#) 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** [199]f 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** [246]f 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** [246]f 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 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”** [314]f 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 \[87\]](#) of the JDO Overview. For a definitive reference, consult the **JDO spe-**

cification.

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** [235] 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>
```

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*** [246] 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 `reversetutorial_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 5.4, “Enhancement”** [229] of 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” \[310\]](#) 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” \[314\]](#) 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 5.5, “Auto-Generating Classes from a Schema” \[230\]](#) 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 7.3, “Mapping File XML Format” \[253\]](#) 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 5.4, “Enhancement” \[229\]](#) 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 [87]** 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 "Surpassing Hubble" in any of its magazines:

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

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. 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`). 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 6.2 to 7.x

Installation of Kodo into WebLogic requires 3 steps. First ensure that the appropriate JDBC driver is in the system classpath. In WebLogic 6.2.x, this should be in the `startWebLogic.sh/cmd` in your domain directory (`$WL_HOME/config/mydomain`). In WebLogic 7, this file is the `startWLS.sh/cmd` file in the `$WL_HOME/server/bin` directory. Make sure to also add the JDO base jar (`jdo-1.0.1.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-1.0.1.jar` to the system classpath. You can accomplish this by editing `startWebLogic.sh/.cmd`.

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

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

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

Now you can start WebLogic and use the console to deploy Kodo JCA. Browse to your WebLogic admin port (`http://yourhost:7001/console`) and browse to the **Connectors** (**Deployments** -> **Connector Modules**) section 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

Note

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

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

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

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

Now restart SunONE.

SunONE requires a SunONE-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 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 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

Note

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

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

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

From there, remove/move some jars that will cause class conflict issues with either your application or BES. Remove `jakarta-commons-logging-*.jar`. Move `jdo-1.0.1.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-1.0.1.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.

4.4. 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 6.2 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

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. 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 142
 - 1.1. General 143
 - 1.2. Database 145
 - 1.3. Programming with Kodo 147
 - 1.4. How do I ... ? 150
 - 1.5. Common errors 152
 - 1.6. Productivity tools 153
 - 1.7. Performance 154
 - 1.8. Scalability 156
 - 1.9. Application servers 157
 - 1.10. Locking 158
 - 1.11. Transactions 159

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” [21]**.

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 datastore-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 confirm to other standards and specifications, including XML, JTA, JCA, JNDI, JDBC, EJB, JMX, XA, and J2EE.

1.1.1
0. 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 jdossupport@solarmetric.com.

1.1.1
1. 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 jdossupport@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* [459]**

- 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 F, *Development and Runtime Libraries* [491]**

- 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* [118]**.

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” [247]**

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” [248]** for a more detailed description.

1.2.1

0. 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

1. 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 of rows.

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* [101]**.

- 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* [398]**

- 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 11.3, “JDOQL” [90]**. Kodo also offers the ability to directly execute SQL queries against the database. See **Section 13.3, “Direct SQL Execution” [353]**

- 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 7.6, “Version Indicator” \[262\]](#)

- 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 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
3. 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” \[33\]](#).

- 1.3.1
4. 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
5. 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 1.8, “Using Persistent Classes Without Enhancement” \[406\]](#))

6. 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
7. 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 7.2, “Mapping Factory” \[251\]](#) 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
8. 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” \[368\]](#)

- 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 [Chapter 11, *Detach and Attach* \[340\]](#)

- 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” \[214\]](#)

- 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* \[192\]](#)

- 1.4.5.
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 13.3, “Direct SQL Execution” \[353\]](#) This can be useful for migrating your queries from a JDBC application to a JDO application.

- 1.4.6.
How do I issue a query against a Date field?

You need to use a parameter to for the query. For details, see [Section 11.3, “JDOQL” \[90\]](#). 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 ());
```

1.4.7.

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

It depends on several factors. If you invoke `JDOHelper.getObjectId()` on your object during the course of a transaction, you will get the object id as of the beginning of the transaction. If you invoke `JDOHelper.getTransactionObjectId()`, you'll get the most up-to-date object id.

If you're using application identity, then the PK values are populated when you populate them, and the object-id object itself is created upon a call to `makePersistent()`. If you populate the application id PK fields after the `makePersistent()` call, then the return from `JDOHelper.getObjectId()` will be a temporary id, and the return from `JDOHelper.getTransactionObjectId()` will be the most up-to-date id.

If you're using datastore identity, then it depends on how the PK is generated. Basically, unless you're using auto-increment columns (see **Example 5.5, "Auto-Increment Object Ids" [224]**) then Kodo will generate/get the PK upon the call to `makePersistent()`. This is one of the reasons that we don't recommend using auto-increment columns, since that reduces our ability to do batching and a couple other things.

It is guaranteed that `JDOHelper.getTransactionObjectId()` will return the permanent id by the time the current transaction has flushed. This includes manual `KodoPersistenceManager.flush()` invocations as well as automatic incremental flushes or commits.

It is also guaranteed that `JDOHelper.getObjectId()` will return the permanent id by the time the current transaction has committed.

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” \[41\]](#).

- 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. What causes a "MissingResourceException" when using Kodo from my application server?

A number of application servers have bugs with loading a resource from an resource adapter. Be sure you follow any instructions that involve extracting the resource adapter as described in [Section 4.3, “Installing Kodo JCA” \[128\]](#).

- 1.5.4. 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” [373]**

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* [371]**

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” [378]**

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” \[359\]](#)

- 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* \[391\]](#)

- 1.7.5.
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” \[329\]](#)

- 1.7.6.
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.

- 1.7.7.
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” \[200\]](#) and [Section 4.2, “Using a](#)

Third-Party DataSource” [202]

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* [125]**.

- 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* [125]**.

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” \[82\]](#).

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 check for lock violation exceptions at commit time and take the appropriate actions upon failure (such as re-trying the operation).

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* \[81\]](#).

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	167
1.1. Intended Audience	168
2. Configuration	169
2.1. Introduction	170
2.2. Runtime Configuration	171
2.3. Command Line Configuration	172
2.3.1. Code Formatting	172
2.4. Plugin Configuration	173
2.5. JDO Standard Properties	175
2.5.1. javax.jdo.PersistenceManagerFactoryClass	175
2.5.2. javax.jdo.option.ConnectionDriverName	175
2.5.3. javax.jdo.option.ConnectionFactoryName	175
2.5.4. javax.jdo.option.ConnectionFactory2Name	175
2.5.5. javax.jdo.option.ConnectionPassword	176
2.5.6. javax.jdo.option.ConnectionURL	176
2.5.7. javax.jdo.option.ConnectionUserName	176
2.5.8. javax.jdo.option.IgnoreCache	176
2.5.9. javax.jdo.option.Multithreaded	176
2.5.10. javax.jdo.option.NontransactionalRead	177
2.5.11. javax.jdo.option.NontransactionalWrite	177
2.5.12. javax.jdo.option.Optimistic	177
2.5.13. javax.jdo.option.RestoreValues	177
2.5.14. javax.jdo.option.RetainValues	178
2.6. Kodo JDO Properties	179
2.6.1. kodo.AggregateListeners	179
2.6.2. kodo.ClassResolver	179
2.6.3. kodo.ConnectionProperties	179
2.6.4. kodo.ConnectionFactoryProperties	179
2.6.5. kodo.Connection2DriverName	180
2.6.6. kodo.Connection2Password	180
2.6.7. kodo.Connection2URL	180
2.6.8. kodo.Connection2UserName	180
2.6.9. kodo.Connection2Properties	181
2.6.10. kodo.ConnectionFactory2Properties	181
2.6.11. kodo.ConnectionRetainMode	181
2.6.12. kodo.CopyObjectIds	181
2.6.13. kodo.DataCache	182
2.6.14. kodo.DataCacheTimeout	182
2.6.15. kodo.EagerFetchMode	182
2.6.16. kodo.FetchBatchSize	182
2.6.17. kodo.FetchGroups	182
2.6.18. kodo.FilterListeners	183
2.6.19. kodo.FlushBeforeQueries	183
2.6.20. kodo.LicenseKey	183
2.6.21. kodo.ManagedRuntime	183
2.6.22. kodo.ManagementServer	184
2.6.23. kodo.ManagementUI	184
2.6.24. kodo.PersistenceManagerImpl	184
2.6.25. kodo.PersistentClasses	184
2.6.26. kodo.ProxyManager	185
2.6.27. kodo.QueryCache	185
2.6.28. kodo.QueryCompilationCache	185
2.6.29. kodo.RemoteCommitProvider	185

2.6.30. kodo.RestoreMutableValues	186
2.6.31. kodo.RetainValuesInOptimistic	186
2.6.32. kodo.TransactionMode	186
2.6.33. kodo.jdbc.AutoIncrementConstraints	186
2.6.34. kodo.jdbc.ClassIndicator	187
2.6.35. kodo.jdbc.ConnectionDecorators	187
2.6.36. kodo.jdbc.DataSourceMode	187
2.6.37. kodo.jdbc.DBDictionary	187
2.6.38. kodo.jdbc.FetchDirection	188
2.6.39. kodo.jdbc.ForeignKeyConstraints	188
2.6.40. kodo.jdbc.JDBCListeners	188
2.6.41. kodo.jdbc.LRSSize	188
2.6.42. kodo.jdbc.MappingFactory	189
2.6.43. kodo.jdbc.ResultSetType	189
2.6.44. kodo.jdbc.SchemaFactory	189
2.6.45. kodo.jdbc.Schemas	189
2.6.46. kodo.jdbc.SequenceFactory	190
2.6.47. kodo.jdbc.SubclassMapping	190
2.6.48. kodo.jdbc.TransactionIsolation	190
2.6.49. kodo.jdbc.UpdateManager	190
2.6.50. kodo.jdbc.VersionIndicator	191
3. Logging	192
3.1. Logging Channels	193
3.2. Disabling Logging	194
3.3. Log4J	195
3.4. JDK 1.4 java.util.logging	196
3.5. Simple Log	197
3.6. Custom Log	198
4. JDBC	199
4.1. Using the Kodo JDO DataSource	200
4.2. Using a Third-Party DataSource	202
4.2.1. Enlisted Data Sources	202
4.3. Database Support	203
4.3.1. MySQLDictionary parameters	207
4.3.2. OracleDictionary parameters	207
4.4. Configuring the DBDictionary	208
4.5. Accessing Multiple Databases	209
4.6. Setting the Transaction Isolation	210
4.7. Setting the SQL Join Syntax	211
4.8. Configuring the Use of JDBC Connections	212
4.9. Runtime Access to JDBC Connections	214
4.10. Large Result Sets	215
4.11. SQL Statement Ordering & Foreign Keys	218
5. Persistent Classes	219
5.1. Restrictions on Persistent Classes	220
5.2. Object Identity	221
5.2.1. Datastore Identity	221
5.2.2. Application Identity	221
5.2.3. Primary Key Generation	222
5.2.3.1. Sequence Factory	222
5.2.3.2. Auto-Increment	223
5.3. Mutable Second Class Object Fields	226
5.3.1. Restoring Mutable Fields	226
5.3.2. Typing and Ordering	226
5.3.3. Proxies	226
5.3.3.1. Smart Proxies	226
5.3.3.2. Large Result Set Proxies	227
5.3.3.3. Custom Proxies	228
5.4. Enhancement	229

5.5. Auto-Generating Classes from a Schema	230
5.5.1. Customizing Reverse Mapping	232
5.6. Persistent Class List	234
6. Metadata	235
6.1. Generating Default JDO Metadata	236
6.2. JDO Metadata Extensions	237
6.2.1. Relation Extensions	237
6.2.1.1. inverse-owner	237
6.2.1.2. dependent	238
6.2.1.3. element-dependent	238
6.2.1.4. value-dependent	238
6.2.1.5. key-dependent	238
6.2.1.6. type	238
6.2.1.7. element-type	238
6.2.1.8. value-type	239
6.2.1.9. key-type	239
6.2.1.10. lrs	239
6.2.1.11. Example	239
6.2.2. Schema Extensions	239
6.2.2.1. jdbc-size	239
6.2.2.2. jdbc-element-size	239
6.2.2.3. jdbc-value-size	240
6.2.2.4. jdbc-key-size	240
6.2.2.5. jdbc-indexed	240
6.2.2.6. jdbc-element-indexed	240
6.2.2.7. jdbc-value-indexed	240
6.2.2.8. jdbc-key-indexed	240
6.2.2.9. jdbc-ref-indexed	240
6.2.2.10. jdbc-version-ind-indexed	240
6.2.2.11. jdbc-class-ind-indexed	240
6.2.2.12. jdbc-delete-action	241
6.2.2.13. jdbc-element-delete-action	241
6.2.2.14. jdbc-value-delete-action	241
6.2.2.15. jdbc-key-delete-action	241
6.2.2.16. jdbc-ref-delete-action	241
6.2.2.17. Example	242
6.2.3. Object-Relational Mapping Extensions	242
6.2.3.1. jdbc-class-map-name	242
6.2.3.2. jdbc-version-ind-name	242
6.2.3.3. jdbc-class-ind-name	243
6.2.3.4. jdbc-field-map-name	243
6.2.3.5. jdbc-ordered	243
6.2.3.6. jdbc-container-meta	243
6.2.3.7. jdbc-null-ind	243
6.2.3.8. externalizer	243
6.2.3.9. factory	243
6.2.3.10. jdbc-class-ind-value	244
6.2.3.11. Example	244
6.2.4. Miscellaneous Extensions	244
6.2.4.1. fetch-group	244
6.2.4.2. data-cache	244
6.2.4.3. data-cache-timeout	244
6.2.4.4. jdbc-sequence-factory	245
6.2.4.5. jdbc-sequence-name	245
6.2.4.6. jdbc-auto-increment	245
6.2.4.7. Example	245
7. Object-Relational Mapping	246
7.1. Mapping Tool	247
7.1.1. Using the Mapping Tool	248

7.2. Mapping Factory	251
7.2.1. Importing and Exporting Mapping Data	252
7.3. Mapping File XML Format	253
7.4. Mapping Notes	255
7.4.1. Join Attributes	255
7.4.2. Non-Standard Joins	256
7.5. Class Mapping	258
7.5.1. Base Mapping	258
7.5.2. Flat Mapping	259
7.5.3. Vertical Mapping	259
7.5.4. Custom Class Mapping	260
7.6. Version Indicator	262
7.6.1. Version Number Indicator	262
7.6.2. Version Date Indicator	263
7.6.3. State Image Indicator	264
7.6.4. Custom Version Indicator	264
7.7. Class Indicator	266
7.7.1. In-Class-Name Indicator	266
7.7.2. Metadata Value Indicator	267
7.7.3. Custom Class Indicator	268
7.8. Field Mapping	269
7.8.1. Value Mapping	269
7.8.2. Blob Mapping	271
7.8.3. Clob Mapping	272
7.8.4. Byte Array Mapping	274
7.8.5. One-to-One Mapping	275
7.8.6. PC One-to-One Mapping	279
7.8.7. Embedded One-to-One Mapping	280
7.8.8. Collection Mapping	282
7.8.9. Many-to-Many Mapping	284
7.8.10. One-to-Many Mapping	287
7.8.11. PC Collection Mapping	289
7.8.12. Map Mapping	291
7.8.13. N-to-Many Map Mapping	292
7.8.14. Many-to-N Map Mapping	294
7.8.15. Many-to-Many Map Mapping	295
7.8.16. PC Map Mapping	297
7.8.17. N-to-PC Map Mapping	298
7.8.18. PC-to-N Map Mapping	300
7.8.19. PC-to-Many Map Mapping	301
7.8.20. Many-to-PC Map Mapping	302
7.8.21. Custom Field Mapping	304
7.8.22. Externalization	304
8. Schema Information	308
8.1. Schema Reflection	309
8.1.1. Schemas List	309
8.1.2. Schema Factory	309
8.1.3. Schema Generator	310
8.2. Schema Tool	312
8.3. XML Schema Format	314
8.4. The SQLLine Utility	316
9. Runtime Deployment	318
9.1. JDOHelper	319
9.2. KodoHelper	320
9.3. J2EE Deployment	321
9.4. Using Kodo JDO via the Java Connector Architecture	323
9.4.1. Deploying on JBoss 3.0	323
10. JDO Runtime Extensions	324
10.1. KodoPersistenceManagerFactory	325

10.2. KodoPersistenceManager	326
10.2.1. JDO Events	326
10.2.2. PersistenceManager Extension	326
10.3. KodoExtent	327
10.4. KodoQuery	328
10.5. Fetch Configuration	329
10.6. KodoHelper	330
10.7. Query Extensions	331
10.7.1. JDOQL Extensions	331
10.7.1.1. Included Query Extensions	331
10.7.1.2. Developing Custom Query Extensions	333
10.7.1.3. Configuring JDOQL Extensions	333
10.8. Query Aggregates and Projections	334
10.8.1. Query Aggregates	334
10.8.1.1. Using Query Aggregates	334
10.8.1.2. Included Query Aggregates	335
10.8.1.2.1. count	335
10.8.1.2.2. sum	335
10.8.1.2.3. min	335
10.8.1.2.4. max	336
10.8.1.2.5. avg	336
10.8.1.3. Developing Custom Query Aggregates	336
10.8.1.4. Configuring Query Aggregates	336
10.8.2. Query Projections	336
10.8.2.1. Using Query Projections	336
10.8.3. JDOQL Variables in Aggregates and Projections	337
10.8.4. Custom Query Result Class	338
10.8.4.1. Using a Bean Query Result Class	338
10.8.4.2. Using a Generic Query Result Class	338
10.8.4.3. Using a Result Alias	339
11. Detach and Attach	340
11.1. Configuring detachability	341
11.2. Detachable behavior	342
11.3. Detach and Attach Callbacks	344
12. Management and Monitoring	345
12.1. Configuring Local Management / Monitoring	346
12.2. Configuring Remote Management / Monitoring	347
12.3. Configuring Logging for Management / Monitoring	348
13. Enterprise Edition	349
13.1. Integrating with the Transaction Manager	350
13.2. XA Transactions	351
13.2.1. Requirements for Using Kodo with XA Transactions	351
13.2.2. Configuring Kodo to Utilize XA Transactions	351
13.3. Direct SQL Execution	353
13.4. MethodQL	354
14. Performance Pack	355
14.1. SQL Batching	356
14.2. Eager Fetching	357
14.2.1. Configuring Eager Fetching	357
14.2.2. Eager Fetching Considerations	358
14.3. Datastore Cache	359
14.3.1. Overview of Kodo JDO Datastore Caching	359
14.3.2. Kodo JDO Cache Usage	359
14.3.3. Query Caching	362
14.3.4. Tangosol Integration	363
14.3.5. Cache Extension	364
14.3.6. Important Notes	364
14.3.7. Known Issues and Limitations	364
14.4. Remote Event Notification Framework	366

14.4.1. Remote Commit Provider Configuration	366
14.4.2. Customization	367
14.5. Fetch Groups	368
14.5.1. Normal Default Fetch Group Behavior	368
14.5.2. Kodo JDO Fetch Group Behavior	369
14.5.3. Custom Fetch Group Configuration	369
15. Third Party Integration	371
15.1. Overview of Third Party Integration features in Kodo	372
15.2. Apache Ant	373
15.2.1. Common Ant Configuration Options	373
15.2.2. JDO Enhancer Ant Task	375
15.2.3. Application Identity Tool Ant Task	375
15.2.4. JDO Metadata Tool Ant Task	375
15.2.5. Mapping Tool Ant Task	376
15.2.6. Reverse Mapping Tool Ant Task	376
15.2.7. Schema Tool Ant Task	377
15.2.8. Schema Generator Ant Task	377
15.3. XDoclet	378
15.4. Borland JBuilder	383
15.4.1. Installing Kodo Into JBuilder	383
15.4.2. Kodo Configuration from JBuilder	383
15.4.3. Creating and building JDO projects in JBuilder	383
15.4.4. Editing JDO Metadata from JBuilder	384
15.4.5. Editing Mapping Info from JBuilder	384
15.4.6. JBuilder Project Sample	384
15.5. Sun ONE Studio / NetBeans IDE	386
15.5.1. Before Installing Kodo into the IDE	386
15.5.2. Installing Kodo into the IDE	386
15.5.3. Configuring the Kodo Module	386
15.5.4. Kodo Template Wizards	387
15.5.5. JDO DataObject	387
15.5.6. Mapping DataObject	387
15.5.7. Kodo Integration into the Build Process	387
15.5.8. SunONE / NetBeans Sample	387
15.6. Eclipse / WebSphere Studio Integration	389
15.6.1. Installing the Kodo Eclipse Plugin	389
15.6.2. Configuring the Plugin	389
15.6.3. Using Kodo in Eclipse IDEs	390
15.6.4. Eclipse Sample	390
16. Optimization Techniques	391

Chapter 1. Introduction

Kodo JDO is a JDBC-based implementation of the JDO 1.0.1 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” \[172\]](#) 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” \[61\]](#) 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 \[199\]](#) 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 \[199\]](#) 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” [35] 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 \[199\]](#) 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 \[199\]](#) 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 \[199\]](#) 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: `false`

Description: Whether to consider modifications to persistent objects made in the current transaction when evaluating queries. See [Section 7.2, “PersistenceManagerFactory Properties” \[61\]](#) 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” \[61\]](#) 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” \[61\]](#) 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” \[61\]](#) 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” \[61\]](#) 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](#)” [61] 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](#)” [61] 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” \[173\]](#)) for custom `kodo.query.AggregateListeners` to make available to all queries, in addition to the standard set of listeners. See [Section 10.8.1, “Query Aggregates” \[334\]](#) 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” \[173\]](#)) 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 JDO 1.0.1 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” \[173\]](#)) listing properties to configure the driver listed in the `ConnectionDriverName` property described in [Section 2.5.2, “`javax.jdo.option.ConnectionDriverName`” \[175\]](#). See [Chapter 4, *JDBC* \[199\]](#) 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” \[173\]](#)) listing properties for configuration of the `javax.sql.DataSource` in use. See the [Chapter 4, *JDBC* \[199\]](#) 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`” \[175\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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`” \[176\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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`” \[176\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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`” \[176\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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`” \[179\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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`” \[179\]](#), but relates to the alternate connection factory used for non-XA connections. See [Section 13.2, “XA Transactions” \[351\]](#) 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” \[212\]](#) 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” \[173\]](#)) describing the `kodo.datacache.DataCache` to use for data caching. See [Section 14.3.2, “Kodo JDO Cache Usage” \[359\]](#) 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” \[359\]](#) for details.

2.6.15. kodo.EagerFetchMode

Property name: `kodo.EagerFetchMode`

Configuration API: `kodo.conf.JDOConfiguration.getEagerFetchMode`

Resource adaptor config-property: `EagerFetchMode`

Default: `multiple`

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” \[357\]](#) for details.

2.6.16. 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” \[215\]](#) for details.

2.6.17. 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” \[368\]](#) for details.

2.6.18. 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” \[173\]](#)) 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” \[331\]](#) for details.

2.6.19. 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” \[212\]](#) for details.

2.6.20. 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.

2.6.21. 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” \[173\]](#)) 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” \[350\]](#) for details.

2.6.22. kodo.ManagementServer

Property name: `kodo.ManagementServer`

Configuration API: `kodo.conf.JDOConfiguration.getManagementServer`

Resource adaptor config-property: `ManagementServer`

Default: `false`

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) describing the `com.solarmetric.manage.ManagerServer` implementation to use for remote monitoring. For more information, see [Chapter 12, Management and Monitoring \[345\]](#)

2.6.23. kodo.ManagementUI

Property name: `kodo.ManagementUI`

Configuration API: `kodo.conf.JDOConfiguration.getManagementUI`

Resource adaptor config-property: `ManagementUI`

Default: `none`

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) describing the `com.solarmetric.manage.ManagerUI` implementation to use for local monitoring. For more information, see [Chapter 12, Management and Monitoring \[345\]](#)

2.6.24. 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” \[173\]](#)) describing the `javax.jdo.PersistenceManager` type to use at runtime. See [Section 10.2.2, “PersistenceManager Extension” \[326\]](#) for details.

2.6.25. 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 5.6, “Persistent Class List” \[234\]](#) for details.

2.6.26. 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” \[173\]](#)) describing a `kodo.util.ProxyManager` to use for proxying mutable second class objects. See [Section 5.3.3.3, “Custom Proxies” \[228\]](#) for details.

2.6.27. 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” \[182\]](#)) is also enabled, `false` otherwise.

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) describing the `kodo.datacache.QueryCache` implementation to use for caching of queries loaded from the data store. See [Section 14.3.3, “Query Caching” \[362\]](#) for details.

2.6.28. 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” \[173\]](#)) 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>

2.6.29. 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”](#) [173]) describing the `kodo.event.RemoteCommitProvider` implementation to use for distributed event notification. See [Section 14.4.1, “Remote Commit Provider Configuration”](#) [366] Remote Commit Provider Configuration for details.

2.6.30. 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”](#) [177] is not set to `false`.

2.6.31. 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.

2.6.32. 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”](#) [350] Integrating with the Transaction Manager for details.

2.6.33. kodo.jdbc.AutoIncrementConstraints

Property name: `kodo.jdbc.AutoIncrementConstraints`

Configuration API: `kodo.conf.JDBCCConfiguration.getAutoIncrementConstraints`

Resource adaptor config-property: `AutoIncrementConstraints`

Default: `false`

Description: Whether to order SQL to meet dependencies introduced by relations to objects with auto-incrementing primary key values. See [Section 5.2.3.2, “Auto-Increment” \[223\]](#) for details.

2.6.34. kodo.jdbc.ClassIndicator

Property name: `kodo.jdbc.ClassIndicator`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.getClassIndicator`

Resource adaptor config-property: `ClassIndicator`

Default: `in-class-name`

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) describing the default `kodo.jdbc.meta.ClassIndicator` to install on new mappings. See [Section 7.7, “Class Indicator” \[266\]](#) for details.

2.6.35. kodo.jdbc.ConnectionDecorators

Property name: `kodo.jdbc.ConnectionDecorators`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.getConnectionDecorators`

Resource adaptor config-property: `ConnectionDecorators`

Default: `-`

Description: A comma-separated list of plugin strings (see [Section 2.4, “Plugin Configuration” \[173\]](#)) 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.

2.6.36. kodo.jdbc.DataSourceMode

Property name: `kodo.jdbc.DataSourceMode`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.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” \[202\]](#) for details.

2.6.37. kodo.jdbc.DBDictionary

Property name: `kodo.jdbc.DBDictionary`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.getDBDictionary`

Resource adaptor config-property: DBDictionary

Default: Based on the `javax.jdo.option.ConnectionURL` (see [Section 2.5.6, “javax.jdo.option.ConnectionURL” \[176\]](#)) and `javax.jdo.option.ConnectionDriverName` (see [Section 2.5.2, “javax.jdo.option.ConnectionDriverName” \[175\]](#)).

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) 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” \[203\]](#) for details.

2.6.38. 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” \[215\]](#) for details.

2.6.39. 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” \[218\]](#) for details.

2.6.40. kodo.jdbc.JDBCListeners

Property name: `kodo.jdbc.JDBCListeners`

Configuration API: `kodo.conf.JDBCCConfiguration.getJDBCListeners`

Resource adaptor config-property: JDBCListeners

Default: -

Description: A comma-separated list of plugin strings (see [Section 2.4, “Plugin Configuration” \[173\]](#)) 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.

2.6.41. 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”](#) [215] for details.

2.6.42. 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”](#) [173]) describing the `kodo.jdbc.meta.MappingFactory` to use to store and retrieve object-relational mapping information for your persistent classes. See [Section 7.2, “Mapping Factory”](#) [251] for details.

2.6.43. kodo.jdbc.ResultSetType

Property name: `kodo.jdbc.ResultSetType`

Configuration API: `kodo.conf.JDBCCConfiguration.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”](#) [215] for details.

2.6.44. kodo.jdbc.SchemaFactory

Property name: `kodo.jdbc.SchemaFactory`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.getSchemaFactory`

Resource adaptor config-property: SchemaFactory

Default: native

Description: A plugin string (see [Section 2.4, “Plugin Configuration”](#) [173]) describing the `kodo.jdbc.schema.SchemaFactory` to use to store and retrieve information about the database schema. See [Section 8.1.2, “Schema Factory”](#) [309] for details.

2.6.45. kodo.jdbc.Schemas

Property name: `kodo.jdbc.Schemas`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.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”](#) [309] for details.

2.6.46. 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”](#) [173]) describing the `kodo.jdbc.schema.SequenceFactory` to use to generate datastore identity values. See [Section 5.2.3.1, “Sequence Factory”](#) [222] for details.

2.6.47. kodo.jdbc.SubclassMapping

Property name: `kodo.jdbc.SubclassMapping`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSubclassMapping`

Resource adaptor config-property: `SubclassMapping`

Default: `flat`

Description: A plugin string (see [Section 2.4, “Plugin Configuration”](#) [173]) describing the default `kodo.jdbc.meta.ClassMapping` to install on new subclass mappings. See [Section 7.5, “Class Mapping”](#) [258] for details.

2.6.48. kodo.jdbc.TransactionIsolation

Property name: `kodo.jdbc.TransactionIsolation`

Configuration API: `kodo.conf.JDBCConfiguration.getTransactionIsolation`

Resource adaptor config-property: `TransactionIsolation`

Default: `default`

Description: The JDBC transaction isolation level to use. See [Section 4.6, “Setting the Transaction Isolation”](#) [210] for details.

2.6.49. kodo.jdbc.UpdateManager

Property name: `kodo.jdbc.UpdateManager`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.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.

2.6.50. kodo.jdbc.VersionIndicator

Property name: `kodo.jdbc.VersionIndicator`

Configuration API: `kodo.jdbc.conf.JDBCCConfiguration.getVersionIndicator`

Resource adaptor config-property: `VersionIndicator`

Default: `version-number`

Description: A plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)) describing the default `kodo.jdbc.meta.VersionIndicator` to install on new mappings. See [Section 7.6, “Version Indicator” \[262\]](#) 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 will 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.

Kodo JDO uses the **Apache Jakarta Commons Logging** thin library for issuing log messages. The Commons Logging libraries act as a wrapper around a number of popular logging APIs, including the **Jakarta Log4J** project, and the native **java.util.logging** package in JDK 1.4. If neither of these libraries are available, then logging will fall back to using simple console 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.

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 5.4, “Enhancement” \[229\]](#)) or `kodo.Metadata` for the mapping tool (see [Section 7.1, “Mapping Tool” \[247\]](#)). 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.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.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.Schema`: Details about operations on the database schema.
- `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.

3.2. Disabling Logging

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

```
java -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.NoOpLog mypkg.MyClass
```

However, disabling logging permanently will cause all error messages to be consumed. So, we recommend using one of the more sophisticated mechanisms described below.

Note

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

3.3. Log4J

When Apache Log4J jars are present, the Commons Logging package will use Log4J by default. In a standalone application, logging levels are controlled by a resource named `log4j.properties`, which should be available as a top-level resource (either at the top level of a jar file, or in the root of one of the CLASSPATH 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 3.1. Standard Logging

```
log4j.rootCategory=WARN, console
log4j.category.kodo.Tool=INFO
log4j.category.kodo.Runtime=INFO
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
```

Example 3.2. Quiet Logging

```
log4j.rootCategory=ERROR, console
log4j.category.kodo.Tool=ERROR
log4j.category.kodo.Runtime=ERROR
log4j.category.kodo.DataCache=ERROR
log4j.category.kodo.Metadata=ERROR
log4j.category.kodo.Enhance=ERROR
log4j.category.kodo.Query=ERROR
log4j.category.kodo.jdbc.SQL=ERROR
log4j.category.kodo.jdbc.JDBC=ERROR
log4j.category.kodo.jdbc.Schema=ERROR

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

Example 3.3. Verbose Logging

```
log4j.rootCategory=TRACE, console
log4j.category.kodo.Tool=TRACE
log4j.category.kodo.Runtime=TRACE
log4j.category.kodo.DataCache=TRACE
log4j.category.kodo.Metadata=TRACE
log4j.category.kodo.Enhance=TRACE
log4j.category.kodo.Query=TRACE
log4j.category.kodo.jdbc.SQL=TRACE
log4j.category.kodo.jdbc.JDBC=TRACE
log4j.category.kodo.jdbc.Schema=TRACE

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

3.4. JDK 1.4 java.util.logging

When using JDK 1.4 or higher, 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 3.4. 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.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
```

3.5. Simple Log

When a version of Java lower than 1.4 is being used, and Log4J libraries are not located in the CLASSPATH, then the commons logging package will fall back to using its built-in simple logging system, using the class `org.apache.commons.logging.impl.SimpleLog`. This system is controlled by the properties in the `simplelog.properties` resource, or else by various system properties, as specified at the [SimpleLog Javadoc](#).

```
java -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog mypkg.MyClass
```

Example 3.5. Example Simple Log Properties

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

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

# set the default logging level for all Kodo logs to "info"
org.apache.commons.logging.simplelog.log.kodo.Tool=info
org.apache.commons.logging.simplelog.log.kodo.Runtime=info
org.apache.commons.logging.simplelog.log.kodo.DataCache=info
org.apache.commons.logging.simplelog.log.kodo.Metadata=info
org.apache.commons.logging.simplelog.log.kodo.Enhance=info
org.apache.commons.logging.simplelog.log.kodo.Query=info
org.apache.commons.logging.simplelog.log.kodo.jdbc.SQL=info
org.apache.commons.logging.simplelog.log.kodo.jdbc.JDBC=info
org.apache.commons.logging.simplelog.log.kodo.jdbc.Schema=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 **org.apache.commons.logging.LogFactory** class. To plug your implementation into the runtime, set the `org.apache.commons.logging.LogFactory` system property to the full name of your log factory class. We present a custom log class example below.

Example 3.6. Custom Logging Class

```
import org.apache.commons.logging.*;
import org.apache.commons.logging.impl.*;

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

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

Chapter 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” \[173\]](#).

- **`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:
 - `MaxActive`: The maximum number of database connections in use at one time. Defaults to 8.
 - `MaxIdle`: The maximum number of idle database connections to keep in the pool. Defaults to 8.
 - `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.
 - `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 set-

tings 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
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” [201]**

- **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 0 to immediately throw an exception. Set to 1 to block until a connection is available or the maximum wait time is exceeded. Set to 2 to automatically grow the pool. Defaults to 1 (block).
- **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.

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 70 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”](#) [350]. 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”](#) [351].

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.
- `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:

- `CreatePrimaryKeys`: If `false`, then do not create database primary keys for identifiers. Defaults to `true`.
- `SchemaCase`: Set this property to "upper", "lower", "preserve", or "default", depending on whether the database will internally represent schema names in upper-case, lower-case, or if it handles case-sensitive names. The default setting is the default case-handling for the database, as reported by the database's metadata.
- `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`.

- `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`.
- `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.
- `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.
- `SystemTables`: Comma-separated list of table names that should be excluded when reflecting on the database. This list will be added to the default list of system tables, which is database-dependent.
- `SystemSchemas`: Comma-separated list of schema names that should be excluded when reflecting on the database. This list will be added to the default list of system schemas, which is database-dependent.
- `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"`.
- `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.
- `StorageLimitationsFatal`: If true, then any data truncation/rounding that is performed by the dictionary in order to store the value in the database will be treated as a fatal error, rather than just issuing a warning.
- `SupportsDeferredConstraints`: If true, then constraints will be generated as being deferrable when appropriate.
- `SupportsForeignKeys`: If true, then the database supports foreign keys.
- `SupportsMultipleNontransactionalResultSets`: If true, then a nontransactional connection is capable of having multiple open `ResultSet` instances.
- `SupportsSelectForUpdate`: If true, then the database supports `SELECT` statements with a pessimistic locking clause.
- `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.
- `MaxIndexesPerTable`: The maximum number of indexes that can be created for a table.
- `MaxKeyNameLength`: The maximum length of any primary or foreign key name.
- `UseGetBestRowIdentifierForPrimaryKeys`: If true, then metadata queries will use `DatabaseMetaData.getBestRowIdentifier` to obtain information about primary keys, rather than `DatabaseMetaData.getPrimaryKeys`.
- `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`.
- `UseStringsForClobs`: If true, then `ResultSet.getString` will be used to obtain blob data rather than `ResultSet.getCharacterStream`.
- `ValidationSQL`: The default SQL that is used by validate that a connection is still in a valid state. For example, "SELECT SYSDATE FROM DUAL" for Oracle.
- `ReservedWords`: A comma-separated list of words that are reserved by this database (in addition to the ones that are reported by the metadata).
- `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.
- `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.
- `InitializationSQL`: A piece of SQL to issue against the database whenever a connection is retrieved from the Data-Source. Defaults to null.
- `CatalogSeparator`: The character the database uses to delimit between the schema name and the table name. This is typically " . ", and defaults to the value returned from the JDBC driver's database metadata.

- `UseSchemaName`: If `false`, then avoid including the schema name in table name references. Defaults to `true`.
- `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.JAVAOBJECT`. 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`.
- `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.

4.3.1. MySQLDictionary parameters

The `mysql` dictionary also understands the following property:

- `TableType`: The MySQL table type to use when creating tables.

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 `AutoIncrementSequenceName` parameter will hold the value of the new primary key for rows that have been inserted. For more details on auto-increment support, see [Example 5.5, “Auto-Increment Object Ids” \[224\]](#)
- `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>"`.

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
import kodo.jdbc.sql.*;        // for Join

...

KodoQuery kq = (KodoQuery) pm.newQuery (MyClass.class, "foo = bar");
((JDBCFetchConfiguration) kq.getFetchConfiguration ()).
    setJoinSyntax (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” \[325\]](#) 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

	FlushBeforeQueries = false	FlushBeforeQueries = true	FlushBeforeQueries = with-connection; ConnectionRetainMode = on-demand	FlushBeforeQueries = with-connection; ConnectionRetainMode = transaction or persistence-manager
IgnoreCache = true	no flush	no flush	no flush	no flush
IgnoreCache = false; no tx active	no flush	no flush	no flush	no flush
IgnoreCache = false; datastore tx active	no flush	flush	flush	flush
IgnoreCache = false; optimistic tx active	no flush	flush	no flush unless 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.query.*;
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 query
KodoQuery kq = (KodoQuery) pm.newQuery (MyClass.class, "foo = bar");
kq.getFetchConfiguration ().setFlushBeforeQueries (KodoQuery.FLUSH_TRUE);
Collection results = (Collection) kq.execute ();
```

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 un-

known. 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
```

Chapter 5. Persistent Classes

Persistent class basics are covered in [Chapter 4, *PersistenceCapable* \[32\]](#) 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” \[35\]](#). 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* \[246\]](#) in object-relational mapping 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 (see [Section 4.5, “JDO Identity” \[40\]](#) 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” \[172\]](#)), including code formatting flags described in [Section 2.3.1, “Code Formatting” \[172\]](#). It also accepts the following arguments:

- `-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. Primary Key Generation

Kodo supports two styles of primary key generation: sequence factories and auto-identity columns.

5.2.3.1. 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 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 `DBSequenceFactory` 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.
- `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}")
```

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.setPriaryKey (gen.getNext ().longValue ());
pm.makePersistent (pc1);

MyPCClass pc2 = new MyPCClass ();
pc2.setPriaryKey (gen.getNext ().longValue ());
pm.makePersistent (pc2);

pm.currentTransaction ().commit ();
```

5.2.3.2. 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 iden-

tity, 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. Persistent-new objects using auto-increment columns for their primary keys will not be assigned their final JDO oid until after the persistence manager flushes to the database, as demonstrated by [this example](#).

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>
```

When you use auto-incrementing primary key columns under either datastore or application identity, Kodo is required to do additional work when it flushes your objects to the database. Kodo must ensure that database rows with auto-incrementing primary keys are flushed before rows that need to store those primary keys as part of a foreign key relation (actual or logical). Therefore, if any of your persistent types have relations to classes with auto-incrementing primary key columns, you must set the following configuration property:

```
kodo.jdbc.AutoIncrementConstraints: true
```

If you do not set the above property, Kodo may create invalid database relations.

Example 5.5. Auto-Increment Object Ids

This example shows when you can get the permanent object id for an object with an auto-incrementing primary key.

```
Person person = new Person ();
person.setFirstName ("Abe");
person.setLastName ("White");

// when you initially make the object persistent, it has a temporary oid
pm.currentTransaction ().begin ();
pm.makePersistent (person);
Object oid = pm.getObjectId (person);           // temp oid
Object toid = pm.getTransactionObjectId (person); // same temp oid

// if you need the permanent oid before commit, you can manually flush; this
// doesn't change the standard oid value, which JDO says has to stay the
// same throughout a transaction, but it does make the transactional oid
// value the final oid
((KodoPersistenceManager) pm).flush ();
oid = pm.getObjectId (person);                   // still temp oid
toid = pm.getTransactionObjectId (person);        // permanent oid!

// or you can wait and get the permanent oid value after commit
pm.currentTransaction ().commit ();
oid = pm.getObjectId (person);                   // permanent oid!
oid = pm.getTransactionObjectId (person);        // permanent oid!
```

5.3. 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.

5.3.1. 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`.

5.3.2. 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 5.6. 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...
}
```

5.3.3. 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.

5.3.3.1. 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.

5.3.3.2. 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 5.7. 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 7.8.22, "Externalization" [304]**)
- 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 5.8. 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>
```

5.3.3.3. 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 5.9. Configuring the Proxy Manager

```
kodo.ProxyManager: AssertAllowedType=true, TrackChanges=false
```

5.4. 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 5.10. 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”** [172]), 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 enhanced, class loading errors can occur. In the event of a class load error, simply re-compile and re-enhance the offending classes.

5.5. 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 5.11. 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” [314]**

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 5.12. 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.
- `-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.
- `-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 [246]** 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 5.13. 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.

5.5.1. 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 5.14. 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
```

5.6. 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. This is the only extension needed to mark two-sided relationships; the owning field in the relationship should **not** have this extension. See below for an example.

6.2.1.2. 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.

6.2.1.3. 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.

6.2.1.4. 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.

6.2.1.5. 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.

6.2.1.6. 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).

6.2.1.7. 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 7.8.22, “Externalization” \[304\]](#)) to indicate the element type if a field externalizes to a collection.

6.2.1.8. 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 7.8.22, “Externalization” \[304\]](#)) to indicate the value type if a field externalizes to a map.

6.2.1.9. 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 7.8.22, “Externalization” \[304\]](#)) to indicate the key type if a field externalizes to a map.

6.2.1.10. 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 5.3.3.2, “Large Result Set Proxies” \[227\]](#).

6.2.1.11. 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-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).

6.2.2.6. jdbc-element-indexed

The `jdbc-element-indexed` field extension is equivalent to the `jdbc-indexed` extension, but applies to columns holding collection elements.

6.2.2.7. 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.

6.2.2.8. 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.

6.2.2.9. 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.

6.2.2.10. 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`.

6.2.2.11. 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`.

6.2.2.12. 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” \[218\]](#).

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.

6.2.2.13. 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.

6.2.2.14. 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.

6.2.2.15. 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.

6.2.2.16. 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.

6.2.2.17. 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* \[246\]](#). This section only reviews the extensions that you can use to give hints to the mapping tool (see [Section 7.1, “Mapping Tool” \[247\]](#)) on 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* \[246\]](#). 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` or `vertical`, 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.

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** 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-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 relation can maintain collection order.

6.2.3.6. 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.

6.2.3.7. 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.

6.2.3.8. 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 7.8.22, “Externalization” \[304\]](#) for details.

6.2.3.9. factory

The `factory` field extension names a method to re-create a field value from its externalized form. See [Section 7.8.22,](#)

“Externalization” [304] for details.

6.2.3.10. jdbc-class-ind-value

The `jdbc-class-ind-value` class extension is reserved for classes that use the metadata value class indicator (see [Section 7.7.2, “Metadata Value Indicator” \[267\]](#)). 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.

6.2.3.11. 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. 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” \[368\]](#).

6.2.4.2. 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.

6.2.4.3. 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.

6.2.4.4. 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 2.6.46, “kodo.jdbc.SequenceFactory” \[190\]](#)). For more information on sequence factories, see [Section 5.2.3.1, “Sequence Factory” \[222\]](#).

6.2.4.5. 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.

6.2.4.6. 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. For more information on auto-increment, see [Section 5.2.3.2, “Auto-Increment” \[223\]](#).

6.2.4.7. 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 | 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”** [312]. Unless you are running the mapping tool on all of your persistent types at once, we strongly recommend you use the default `add` schema 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**.
- `-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 option 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 7.3, “Mapping File XML Format” [253]**
- `export`: Export the mapping data for the given classes to an XML file. The XML format used for mapping data is discussed in **Section 7.3, “Mapping File XML Format” [253]**

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
```

7.2. 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” [247]**). 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.

7.2.1. 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 7.8. 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 7.9. 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
```

7.3. 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 7.10. 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 mapping. 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 7.11. 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>
```

7.4. 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.

7.4.1. 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" />
```

7.4.2. 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/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 criteria is that a column in the referenced table must have some constant value. Let's 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.PK2 = '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.

7.5. 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.

7.5.1. 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 7.12. Using a Base Mapping

```
Java class:
public class Magazine
{
    ... class content ...
}

Schema:
<table name="MAGAZINE">
    <column name="JDROID" type="bigint"/>
    <pk column="JDROID"/>
    ... 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="JDROID"/>
    ... 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="JDROID"/>
    </extension>
    ... indicator extensions ...
</class>
```

```
... field metadata      ...  
</class>
```

7.5.2. Flat Mapping

The *flat* class mapping 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 7.13. 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>
```

7.5.3. Vertical 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 7.4.2, “Non-Standard Joins” [256]**.

Example 7.14. 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>
```

7.5.4. 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.

7.6. 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.

7.6.1. 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 7.15. 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>
```

7.6.2. 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 7.16. 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>
```

7.6.3. 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.

The state image indicator has the following attributes:

- `type: state-image`

Example 7.17. 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>
```

7.6.4. 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.

7.7. 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.

7.7.1. 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 7.18. Using a 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>
```

```
</class>
```

7.7.2. 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 7.19. 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 -->
    ... 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>
```

7.7.3. 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.

7.8. 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 7.8.22, “Externalization” \[304\]](#), 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.

7.8.1. Value Mapping

The value mapping represents the direct mapping of a Java primitive, primitive wrapper, 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 7.4.2, “Non-Standard Joins” \[256\]](#).
- `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 7.20. 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 7.21. 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.JDOID="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.JDOID" value="MAG_ID"/>
  </extension>
</field>
... rest of field metadata ...
</class>
```

7.8.2. 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 7.4.2, “Non-Standard Joins” [256]**.
- **`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 7.22. 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>
```

7.8.3. 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.23. 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>
```

7.8.4. 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.24. 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>

```

7.8.5. 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.

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 7.4.2, "Non-Standard Joins" [256]**.
- `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 7.25. 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 7.26. 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.map` 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>
  </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 7.27. 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>
```

7.8.6. 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 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 7.4.2, “Non-Standard Joins” \[256\]](#).
- **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 7.28. 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>
```

7.8.7. 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 7.4.2, “Non-Standard Joins” [256]**.

Example 7.29. 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>
        </jdb
```

```

    </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">
    <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>
    <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>
  </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>

```

7.8.8. Collection Mapping

The collection mapping maps a collection or array of simple values (primitive wrappers 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 7.4.2, “Non-Standard Joins” \[256\]](#).
- `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 7.30. Using a Collection Mapping

```
Java class:
public class Magazine
{
    private Set coverBlurbs;
    ... 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="coverBlurbs">
    <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="coverBlurbs">
    <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="coverBlurbs">
    <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>
```

7.8.9. 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. 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 7.4.2, "Non-Standard Joins" [256]**.
- `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 7.31. 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 7.32. 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>
```

7.8.10. 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.

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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.33. 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 7.34. 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>
```

7.8.11. 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, 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.35. 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>
```

7.8.12. Map Mapping

The map mapping represents a map in which the keys and values are simple types (primitive wrappers 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 7.4.2, “Non-Standard Joins” \[256\]](#).
- **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 7.36. 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>

```

7.8.13. 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 7.4.2, "Non-Standard Joins"](#) [256].

- **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 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 7.37. 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>

```

7.8.14. 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 7.4.2, “Non-Standard Joins” \[256\]](#).
- `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 7.38. 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>
</table>
```

```

    </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>

```

7.8.15. 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 7.4.2, "Non-Standard Joins" [256]**.

- **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 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 7.39. 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>

```

7.8.16. 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, 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.40. 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>

```

7.8.17. N-to-PC Map Mapping

The n-to-PC map mapping represents a map whose keys are a simple type (primitive wrapper, 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, 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 7.4.2, “Non-Standard Joins” \[256\]](#).

- **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 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 7.41. 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.JDOID" value="MAG_ID"/>
    </extension>
  </field>
  ... rest of field metadata ...
</class>

```

7.8.18. 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, string). This is the default mapping for maps with simple values and user-defined interface keys, 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 7.42. 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>
```

7.8.19. 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 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 7.4.2, "Non-Standard Joins" [256]**.
- **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 7.43. Using a PC-to-Many Map Mapping

```

Java class:

public class Magazine
{
    private Map chapterMarkers;
    ... class content ...
}

Schema:

<table name="MAGAZINE">
    <column name="JDROID" type="bigint"/>
    <pk column="JDROID"/>
    ... 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="JDROID"/>
    </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.TITLE="CHAPT_TITLE" table="CHAPTERMARKERS"
            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="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.JDROID" value="MAG_ID"/>
        </extension>
    </field>
    ... rest of field metadata ...
</class>

```

7.8.20. 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 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 7.4.2, “Non-Standard Joins” [256]**.
- `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 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>
```

7.8.21. 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.

7.8.22. 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><extension vendor-name="kodo" key="externalizer" value="toString"/></code>
<code>public String CustomType.toString(PersistenceManager pm)</code>	<code><extension vendor-name="kodo" key="externalizer" value="toString"/></code>
<code>public static String AnyClass.toString(CustomType ct)</code>	<code><extension vendor-name="kodo" key="externalizer" value="AnyClass.toString"/></code>
<code>public static String AnyClass.toString(CustomType ct, PersistenceManager pm)</code>	<code><extension vendor-name="kodo" key="externalizer" value="AnyClass.toString"/></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><extension vendor-name="kodo" key="factory" value="fromString"/></code>
<code>public static CustomType CustomType.fromString(String str, PersistenceManager pm)</code>	<code><extension vendor-name="kodo" key="factory" value="fromString"/></code>
<code>public static CustomType AnyClass.fromString(String str)</code>	<code><extension vendor-name="kodo" key="factory" value="AnyClass.fromString"/></code>
<code>public static CustomType AnyClass.fromString(String str, PersistenceManager pm)</code>	<code><extension vendor-name="kodo" key="factory" value="AnyClass.fromString"/></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 7.45. 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. Declare the parameter to be the actual field type, and pass in a value of that 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 7.46. 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);
q.declareParameters ("java.net.URL u");
q.setFilter ("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 ();
```

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, however, that the dynamic schema factory cannot auto-detect foreign keys, column defaults, non-nullable columns, and other database information.
- **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” \[314\]](#). 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`.

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.
- `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 '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>
```

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 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 subsequence 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 into your J2EE applications in a number of ways. The simplest is just to include the Kodo libraries in a J2EE EAR file and call Kodo JDO directly from there using the standard `JDOHelper.getPersistenceManagerFactory` described in the previous section. This will be problematic, however, if you want Kodo JDO to share resources with other beans or if you have multiple beans deployed that need to use the same Kodo JDO resources.

The second way to integrate Kodo JDO is to configure and bind an instance of the `PersistenceManagerFactory` into JNDI. Although the mechanism by which you do this is up to you, the most common way is to create a startup class according to your application server's documentation that will create a factory and then bind it in JNDI. For example, in WebLogic you could write a startup class as follows:

Example 9.4. Binding a PersistenceManagerFactory into JNDI via a WebLogic Startup Class

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

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;

/**
 * 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);

        InitialContext ic = new InitialContext ();
        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 9.5. Looking up the PersistenceManagerFactory in JNDI

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

The third way to deploy Kodo JDO in an application server is to utilize the Java Connector Architecture features of Kodo JDO as described in the next section.

9.4. Using Kodo JDO via the Java Connector Architecture

An introduction to JCA in **Javaworld** gives a good overview of the Java Connector Architecture:

The JCA defines a standard set of interfaces that allows connectors to integrate with compliant application servers seamlessly. At the same time, another standard set of interfaces allows clients (or applications hosted by the application server) to use the connectors in a uniform way. Thus with JCA, connectors are portable across application servers, and clients are portable across connectors.

—Tarak Modi

Kodo JDO can be deployed through JCA in any JCA-compliant application server. The next section demonstrates how to deploy Kodo on JBoss.

For information on deploying to other JCA-compliant application servers see **Section 4.3, “Installing Kodo JCA” [128]** in the J2EE Tutorial.

9.4.1. Deploying on JBoss 3.0

JBoss 3.0 will automatically deploy the `kodo.rar` file (located in this distribution's `jca` directory) when it is placed in JBoss' deploy directory. However, you will need to configure the parameters of the resource adaptor by creating a `kodo-service.xml` file in the JBoss deploy directory. A template `kodo-service.xml` is provided with this distribution in the `jca` directory.

Note that while JBoss versions lower than 3.0 support the JCA, they provide no way to configure the `ManagedConnectionFactoryProperties`, which is the primary means of configuring Kodo JDO as a service in a managed environment. If you want to deploy Kodo JDO to JBoss 2.4.4 or lower, you will need to extract the `ra.xml` from the `kodo.rar` file, change the default values settings to match your configuration, and then repack the files back into the `kodo.rar` file.

To deploy to JBoss, you must do the following:

- Copy `jca/kodo.rar` to the deploy directory of your JBoss installation.
- Edit `jca/kodo-service.xml` to set up the configuration for your environment. In particular, you must set the `LicenseKey` property to your license key, and you will probably need to change the `ConnectionDriverName`, `ConnectionURL`, `ConnectionUserName`, `ConnectionPassword`, and `DBDictionaryClass` properties.
- Copy the updated `jca/kodo-service.xml` to the deploy directory of your JBoss installation.

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 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. 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 property:

- `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.

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 have many configuration parameters and functions beyond the standard `javax.jdo.Query`, including **aggregates and projections**, a JDO 2.0 preview feature. Access to these extensions 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** and **custom fetch groups**. 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` 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. See the previous chapter for a discussion of **direct SQL queries** and **custom query execution**.

10.7.1. JDOQL Extensions

JDOQL extensions are custom methods that you can use in your query filter string. Kodo JDO provides some built-in query extensions, and you can develop your own custom extensions as needed. You preface all JDOQL extensions with `ext :` in your JDOQL query. For example, the following example uses Kodo's built-in `endsWith` extension to search for cities whose name ends with "ford" (e.g. Hartford):

Example 10.1. Basic Query Extension

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

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

Example 10.2. Chaining Query Extensions

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

Finally, when using query extensions you must be aware that some SQL-specific extensions can only execute against the database, and cannot be used for in-memory queries (recall that JDO executes queries in-memory when you supply a candidate collection rather than an extent/class, or when you set the `IgnoreCache` and `FlushBeforeQueries` properties to `false` and you execute a query within a transaction in which you've modified some persistent objects). In the list of built-in query extensions below, you can assume each extension can be executed both in-memory and against the database unless its description says otherwise.

10.7.1.1. Included Query Extensions

Kodo includes several default query extensions to enhance the power of JDOQL. Some of these extensions are being considered for inclusion in the official JDO 2.0 specification, so they may become standard in the future.

- `containsKey`: Tests whether the target map contains the given argument in its key set. The argument may be a JDOQL variable, parameter, or constant.

```
query.declareVariables ("Employee emp");
query.setFilter ("employeeMap.ext:containsKey (emp) "
    + "&& emp.salary > 50000");
```

Note

This query extension is available free of charge in all editions of Kodo JDO.

- `containsValue`: Tests whether the target map contains the given argument in its value collection. The argument may be a JDOQL variable, parameter, or constant.

```
query.declareVariables ("Company comp");
query.setFilter ("companyMap.ext:containsValue (comp) "
    + "&& comp.address.city == \"Houston\"");
```

Note

This query extension is available free of charge in all editions of Kodo JDO.

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

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

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

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

- `matches`: Tests if the target string matches the given regular expression. Equivalent to Java 1.4's `String.matches ()` method, though only a subset of the standard regular expression syntax is supported:
 - `.*`: Matches any 0 or more characters.
 - `.`: Matches any 1 character.
 - `(?i)`: Placing this token in the regular expression string means the expression will ignore case when matching.

This extension replaces `stringContains` and `wildcardMatch`, which are now deprecated.

```
query.setFilter ("stringField.ext:matches (\".*jdo.(?i)\")");
```

- `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\\"");
```

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

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

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

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

10.7.1.2. Developing Custom Query 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 query extensions with Kodo:

- *Registration by properties*: You can register custom query 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 query extensions for an individual query through the `KodoQuery.registerListener` method. You might use per-query registration for very specific extensions that do not apply globally.

10.8. Query Aggregates and Projections

Aggregates and projections allow a JDO application to efficiently work with an aggregation or subset of the data that would be returned from a standard JDO query. The APIs for manipulating aggregations and projections are not found in the standard `javax.jdo.Query` interface, so you must cast your queries to to **`kodo.query.KodoQuery`** in order to use them. These APIs are under consideration for version 2.0 of the JDO specification, however, so they may become standard in the future.

10.8.1. Query Aggregates

Aggregates perform simple statistical functions on the set of values returned by a query. While aggregate functions can always be computed by performing the calculations in memory against the results of a query, it is often inefficient to do so, since it can potentially draw down large amounts of data from the datastore. For example, datastore aggregate functions can be used to quickly calculate the average of millions of fields without having to instantiate each of the objects in the JVM.

10.8.1.1. Using Query Aggregates

Use the **`KodoQuery.setResult`** method to ask for aggregate data. The value passed in to the `setResult` method will be one or more aggregate functions of the form "function-name(parameter)". The given parameter can be any field traversal path or mathematical expression that would be legal as a value in a JDOQL filter. Multiple aggregates are separated by commas. See the section on **built-in aggregate functions** for details and examples.

When using aggregates, the return value of the `Query.execute` method will be either `Object` (if only a single aggregate is requested), or `Object[]` (if multiple aggregates are requested). The types of the returned objects depend on the aggregate functions used. The discussion of built-in aggregates below includes the return type for each aggregate. You can also override the default return type using a custom result class, which we describe **later**.

Example 10.3. Basic Query Aggregate

```
KodoQuery q = (KodoQuery) pm.newQuery (Product.class);
q.setResult ("avg(cost)");
Number averageCost = (Number) q.execute ();
System.out.println ("Average product cost: " + averageCost.doubleValue ());
```

Example 10.4. Multiple Query Aggregates

```
KodoQuery q = (KodoQuery) pm.newQuery (HomeOwner.class);
q.setResult ("count(this), sum(income), min(income), max(income)");
Object[] results = (Object[]) q.execute ();

long totalResidents = ((Number) results[0]).longValue ();
double incomeSum = ((Number) results[1]).doubleValue ();
double minIncome = ((Number) results[2]).doubleValue ();
double maxIncome = ((Number) results[3]).doubleValue ();

System.out.println ("Total residents: " + totalResidents);
System.out.println ("Minimum income: " + minIncome);
System.out.println ("Maximum income: " + maxIncome);
System.out.println ("Average income: " + (incomeSum / totalResidents));
```

If the default return type of an aggregate does not match your needs, you can cast the return type to a specific class in the result

string. When casting to a primitive type, the appropriate primitive wrapper class will be returned from the `Query.execute` method.

Example 10.5. Casting Query Aggregates

```
KodoQuery q = (KodoQuery) pm.newQuery (Student.class);
q.setResult ("(int) avg(height)");
int roundedAverageHeight = ((Integer) q.execute ().intValue ());
```

10.8.1.2. Included Query Aggregates

Kodo includes several default query aggregates that replicate the common aggregate functions of various data stores. If a data store supports additional desired aggregate functions, you can implement them with custom aggregate plugins, described in [Section 10.8.1.3, “Developing Custom Query Aggregates” \[336\]](#)

10.8.1.2.1. count

The `count` function returns the total number of objects returned by a certain query. The default return type of this function is `Long`.

```
// determine the total employees based in new york
KodoQuery q = (KodoQuery) pm.newQuery (Company.class);
query.setFilter ("state == \"New York\"");
query.setResult ("count(this)");
long newYorkCompanies = ((Long) q.execute ().longValue ());
```

10.8.1.2.2. sum

The `sum` function returns the sum total of the parameter. The default return type of the function depends on the field type. If the field is an exact numeric type, the default return type is `Long`. If it is a floating point type `T`, the default return type is `T`. If you are taking the sum of a mathematical expression, the return type is `Number`.

```
// determine the sum total of all sales made in 2003
KodoQuery q = (KodoQuery) pm.newQuery (Sales.class);
query.setFilter ("year == 2003");
query.setResult ("sum(amount)");
double yearSales = ((Double) q.execute ().doubleValue ());
```

10.8.1.2.3. min

The `min` function returns the minimum value of the parameter. The default return type of this function is the type of the field it is applied to. If you apply this function to a mathematical expression, the default return type is `Number`.

```
// determine the shortest of all employees
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
query.setResult ("min(height)");
float shortest = ((Float) q.execute ().floatValue ());
```

10.8.1.2.4. max

The `max` function returns the maximum value of the parameter. The default return type of this function is the type of the field it is applied to. If you apply this function to a mathematical expression, the default return type is `Number`.

```
// determine the maximum salary of all the employees
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
query.setResult ("max(salary)");
double richest = ((Double) q.execute ()).doubleValue ();
```

10.8.1.2.5. avg

The `avg` function returns the average value of the parameter. The default return type of this function is the type of the field it is applied to. If you apply this function to a mathematical expression, the default return type is `Number`.

```
// calculate the average sales for all employees based in california
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
query.setFilter ("company.state == \"California\"");
query.setResult ("avg(sales.amount)");
double avgSale = ((Double) q.execute ()).doubleValue ();
```

10.8.1.3. Developing Custom Query Aggregates

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 strings, prefix the function name with `ext :` to identify it as an extension to the standard set of aggregates.

10.8.1.4. 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.registerListener` method. You might use per-query registration for very specific aggregates that do not apply globally.

10.8.2. Query Projections

A projection is a set of field values that are components of the object graph that would result from a query. Projections can be useful for efficiently obtaining simple field values or relations from the candidate class without having to instantiate each instance.

10.8.2.1. Using Query Projections

Use the `KodoQuery.setResult` method to ask for projection data. The string you pass to the method will be one or more field names or relation traversal strings separated by commas.

When using projections, the `execute` method will return a collection whose elements are either instances of the value of the field being projected (or the wrapper class for primitive types), or arrays of type `java.lang.Object` whose elements will be instances of the field's class. You can also use a custom class to encapsulate the result data, as described in [Section 10.8.4, “Custom Query Result Class” \[338\]](#) in Custom Result Classes.

Example 10.6. Basic Query Projections

```
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
q.setResult ("company.name");
Collection results = (Collection) q.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    System.out.println ("Company name: " + itr.next ());
q.close (results);
```

Example 10.7. Multiple Query Projections

```
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
q.setResult ("company.name, dateOfBirth, address");
Collection results = (Collection) q.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] result = (Object[]) itr.next ();
    String companyName = (String) result[0];
    Date dateOfBirth = (Date) result[1];
    Address address = (Address) result[2];
    // process this data...
}
q.close (results);
```

10.8.3. JDOQL Variables in Aggregates and Projections

In JDOQL filter strings, you use variables to traverse collection and map fields. Each variable is typically constrained by including it in a `contains` clause of the filter string:

```
Query q = pm.newQuery (Company.class);
q.declareVariables ("Product prod");
q.setFilter ("products.contains (prod) && prod.price < 10.0");
```

You can also use variables in the result string in your aggregates and projections, though the `contains` clause stays in the filter portion of the query. For example, the query below finds minimum price of any product in the Acme company.

```
KodoQuery q = (KodoQuery) pm.newQuery (Company.class);
q.declareVariables ("Product prod");
q.setResult ("min(prod.price)");
q.setFilter ("name == \"Acme\" && products.contains (prod)");
double price = ((Number) q.execute ().doubleValue ());
```

Similarly, to retrieve all the products of the Acme company without having to instantiate the company and traverse the relation, you can use a variable projection:

```
KodoQuery q = (KodoQuery) pm.newQuery (Company.class);
q.declareVariables ("Product prod");
q.setResult ("prod");
q.setFilter ("name == \"Acme\" && products.contains (prod)");
Collection products = (Collection) q.execute ();
```

10.8.4. Custom Query Result Class

When using aggregate or projection queries, it is possible to specify that the resulting value (or collection of values) will be held in a specific custom class. Use the `KodoQuery.setResultClass` method to declare the class that will be instantiated to hold the results of the query.

10.8.4.1. Using a Bean Query Result Class

The class that is the parameter to `setResultClass` can define JavaBean setter methods for field values, and these setters will be invoked for each field that results from the aggregate or projection query. The custom result class must be public (or otherwise configured to be able to be instantiated via reflection), and must have a no-args constructor.

Example 10.8. Using a Result Class Bean

```
public static class SalesData
{
    private Date salesDate;
    private double amount;

    public void setSalesDate (Date salesDate)
    {
        this.salesDate = salesDate;
    }

    public Date getSalesDate ()
    {
        return this.salesDate;
    }

    public void setAmount (double amount)
    {
        this.amount = amount;
    }

    public double getAmount ()
    {
        return this.amount;
    }
}

KodoQuery q = (KodoQuery) pm.newQuery (Sales.class);
q.setResult ("amount, salesDate");
q.setResultClass (SalesData.class);
Collection data = (Collection) q.execute ();
for (Iterator itr = data.iterator (); itr.hasNext ();)
{
    SalesData item = (SalesData) itr.next ();
    System.out.println ("Date: " + item.getSalesDate ()
        + " amount: " + item.getAmount ());
}
```

10.8.4.2. Using a Generic Query Result Class

Whenever the specified result class does not contain a setter method for the aggregate or projection name, Kodo will try to detect a custom method named `put` that takes two `java.lang.Object` arguments. If found, Kodo will invoke that method with the aggregate or projection name as the first argument, and the value as the second argument. A custom result class can thus define a generic `put(Object, Object)` for use with any aggregate or projection queries. Also, this contract is suitable for use with any implementation of the `java.util.Map` interface which defines a `Map.put` method.

Example 10.9. Using a Generic Result Class

```
KodoQuery q = (KodoQuery) pm.newQuery (Employee.class);
q.setResult ("firstName, lastName, company.name");
q.setResultClass (HashMap.class);
Collection data = (Collection) q.execute ();
for (Iterator itr = data.iterator (); itr.hasNext ();)
{
    HashMap item = (HashMap) itr.next ();
    System.out.println ("Name: "
        + item.get ("firstName") + " " + item.get ("lastName")
        + " Company: " + item.get ("company.name"));
}
```

10.8.4.3. Using a Result Alias

It is possible to declare aliases in the `setResult` invocation for both aggregates and projections. These aliases are of the form "field or aggregate function as alias". The alias will be used for the result class' bean setter, or for the name of the `put` key in generic result class handling.

Example 10.10. Using a Result Alias

```
public static class SalesStatistics
{
    private int totalSales;
    private double averageSales;

    public void setTotalSales (int totalSales)
    {
        this.totalSales = totalSales;
    }

    public int getTotalSales ()
    {
        return this.totalSales;
    }

    public void setAverageSales (double averageSales)
    {
        this.averageSales = averageSales;
    }

    public double getAverageSales ()
    {
        return this.averageSales;
    }
}

KodoQuery q = (KodoQuery) pm.newQuery (Sales.class);
q.setResult ("count(this) as totalSales, avg(amount) as averageSales");
q.setResultClass (SalesStatistics.class);
SalesStatistics stats = (SalesStatistics) q.execute ();
System.out.println ("# of sales: " + stats.getTotalSales ());
System.out.println ("Average sale: " + stats.getAverageSales ());
```

Chapter 11. 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 detached 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 2.0 specification, so they may become standard in the future.

11.1. Configuring detachability

In order to be able to detach a persistent instance, the metadata for the class must declare that it is eligible for detachment by 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. The simplest example of the `detachable` extension is:

```
public class DetachExample
    implements java.io.Serializable
{
    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 will add additional fields to the enhanced version of the class. One of these fields is of type `java.lang.Object`, and holds an object that refers to the class' state, so that version checking can be done when the object is re-attached. If the persistent class does not use application identity, then a second `String` field will be added that will be used to store the stringified object id of the instance that was detached.

It is possible to define one or both of these fields and declare them the metadata for your class, which will prevent the enhancer from adding the default fields to the class, and will keep your enhanced class serialization-compatible with the unenhanced version (in case the client tier only has the unenhanced class definition to work with). The `detached-objectid-field` and `detached-state-field` class-level metadata extensions declare these fields. These fields must not be managed by JDO (they must have their `persistence-modifier` attribute set to `"none"`).

```
public class DetachExample
    implements java.io.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>
```

11.2. Detachable behavior

The `KodoPersistenceManager` exposes six methods to support detach and attach functionality:

- `detach(Object ob)`
- `detachAll(Object[] obs)`
- `detachAll(Collection obs)`
- `attach(Object ob)`
- `attachAll(Object[] obs)`
- `attachAll(Collection obs)`

Detaching persistent instances will create unmanaged copies of the instances. The copy mechanism is similar to serialization, with the exception that only fields in the current fetch groups are traversed. Fields that are not in the current fetch groups will be set to the Java default for their type, and will be ignored when the detached instances are later re-attached. It is important to remember that invoking the `detach` method does not modify the persistent instance that is the parameter of the method call; it is the return value from that invocation that is the detached and unmanaged copy.

Example 11.1. Detaching a Single Instance

```
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 ();
```

If relation-type fields need to be detached, then they need to be declared as being in the `default-fetch-group`, or, if the capability is available, be declared in the **custom fetch groups** configuration for the `KodoPersistenceManager` performing the detachment.

Example 11.2. Using Custom Fetch Groups for Detach

```

public class DetachExample
    implements java.io.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 too and apply mods
kpm.currentTransaction ().commit ();

```

When attaching an instance whose representation has changed in the datastore since detachment, a `JDOOptimisticVerificationException` will be thrown upon commit or flush, just as if a normal optimistic conflict was detected. When attaching an instance that represents an object that has been deleted since detaching, or when attaching a detached instance into a `PersistenceManager` that has an earlier version of the object, a `JDOOptimisticVerificationException` will be immediately thrown when attach is invoked. In these cases, the `RollbackOnly` flag will be set on the transaction.

11.3. Detach and Attach Callbacks

Persistent instances can be notified when they are being detached or attached by implementing the `kodo.runtime.PreDetachCallback` `kodo.runtime.PostDetachCallback` `kodo.runtime.PreAttachCallback` and `kodo.runtime.PostAttachCallback` interfaces. These interfaces are analogous to the notification mechanisms provided by the `javax.jdo.InstanceCallbacks` interface (see [Section 4.4, “InstanceCallbacks” \[38\]](#)).

- `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 detach instance that has been attached. It can be used to transfer transient or unmanaged state between the detached instance and the attached instance.

Chapter 12. Management and Monitoring

This release of Kodo includes a preview of a management and monitoring API and set of tools. Note that these capabilities are in preview status, and thus are not as mature as the rest of the product. Also, the APIs and tools are likely to change significantly in minor releases.

The management and monitoring capability allows for both local and remote monitoring of performance related statistics. For example, the connection pool provides statistics on the numbers of active and idle connections, and the data cache provides statistics on the number of cache hits and misses.

The management and monitoring capability can be run in two different ways -- locally, and remotely. An example of its use can be found in the `monitoring` sample described in [Section 1.12, “Management / Monitoring” \[410\]](#)

12.1. Configuring Local Management / Monitoring

To start up a local management and monitoring graphical user interface (GUI), set the `kodo.ManagementUI` property to `gui`. This property is also a plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)), so you can also set it to the full class name of a custom **ManagerUI**. The default management UI implementation creates a window for each `PersistenceManagerFactory` with a unique configuration. The window will contain a number of tabs, each containing a chart with one or more graphs on it.

12.2. Configuring Remote Management / Monitoring

To start up a remote management and monitoring GUI, set the `kodo.ManagementServer` configuration property to `true`. This property is also a plugin string (see [Section 2.4, “Plugin Configuration” \[173\]](#)), so you can also set it to the full class name of a custom **ManagerServer**. The default management server implementation listens for connections from the `remotemanagementtool` on `localhost:1234`. To change the host and port on which the server will listen, the default management server implementation provides a `host` property, and a `port` property. These properties can be set in the plugin string in the standard way. For example, to listen on port 2345:

```
kodo.ManagementServer: true(port=2345)
```

To connect to the server, run the `remotemanagementtool` command. By default, the `remotemanagementtool` attempts to connect on `localhost:1234`. The `remotemanagementtool` command takes a `host` argument and a `port` argument. For example, to connect to port 2345 on a machine named `myhost`:

```
remotemanagementtool -host myhost -port 2345
```

12.3. Configuring Logging for Management / Monitoring

Logging for the management and monitoring API is on the `com.solarmetric.Manage` logging channel.

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 4.1, “Using the Kodo JDO Data-Source” [200]** 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 the documentation for **using the Kodo data source** or **using a third-party data source** 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:/OracleXASource
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. 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 rather than a low-level `ResultSet`. The only requirements are that your SQL selects all primary key columns, that it selects the **class indicator** column if present, and that it does not alias the columns to different names in the select statement. Kodo will take over the execution of the SQL and will scan the result set to determine which fields it can set in the objects it returns to you. If your select statement does not load all the data in the default or currently-configured fetch groups, Kodo will make additional trips to the datastore as needed to load the additional data.

```
// the SQL query language is 'kodo.jdbc.SQL'
Query q = pm.newQuery ("kodo.jdbc.SQL", null);

// set the type of objects that the method returns
q.setClass (Person.class);

// set the filter to the SQL to execute, using named placeholders for any
// parameters you'd like to pass in
q.setFilter ("SELECT * FROM PERSON WHERE FIRSTNAME = first "
    + "AND LASTNAME = last");

// when you declare parameters, the parameter type is the JDBC type from
// java.sql.Types
q.declareParameters ("VARCHAR first, VARCHAR last");

// this executes your SQL and transforms the result set into matching
// objects
Collection results = (Collection) q.execute ("Fred", "Lucas");
```

If you specify a SQL filter that does not begin with the `SELECT` keyword (regardless of case), Kodo will assume that you are executing a stored procedure and use the appropriate JDBC APIs for stored procedure calls rather than SQL selects. Parameter passing is done the same way for either SQL selects or stored procedure calls, as illustrated in the code sample above. Stored procedure OUT parameters are not supported; the stored procedure must return a single result set.

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.

The `StoredProcMain` driver program in the `samples/ormapping` directory of your Kodo distribution demonstrates a working SQL query.

13.4. 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.

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. Additionally, you can buy many Performance Pack features individually. Contact 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 related objects along with the objects being queried. Typically, Kodo JDO has to make a trip to the database whenever a relation is loaded. 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 data store queries (the initial query, plus one for the address of each person returned). With eager fetching, Kodo can reduce this 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's fetch group (or if the address were in the default fetch group). This allows you to control exactly which fields are eagerly fetched.

Eager fetching has three modes:

- **none**: No eager fetching is performed. Related objects are always loaded in a separate select statement.
- **single**: In this mode, 1-1 relations in the configured fetch groups are eagerly fetched when selecting for objects. A left outer join (or inner join, if the 1-1 relations' field metadata sets the `null-value` attribute to `exception`) is used to select the relations' data along with the data for the target objects. This process works recursively, 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.

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`.

- **multiple**: 1-1, 1-many, and many-many relations in the configured fetch groups are all eagerly fetched, along with fields holding collections of simple values.

1-1 relations are selected as described in the **single** mode description above. To-many relations and collection fields, on the other hand, are fetched using a separate select statement for each relation, batched together with the select statement for the target objects. The batched 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, 3 queries will be made. 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 1-1 relations, this process can be recursively applied to the objects in the relations being eagerly fetched. If the `Employee` class had a list of `Projects` in one of the fetch groups being loaded, a single additional select would be batched that would load the projects of each employee of the matching companies.

Using an additional select to load each to-many relation avoids transferring more data than necessary from the database to the application. If outer joins were used instead of separate select statements, each to-many relation 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.

When Kodo knows that it is selecting for a single object only, such as in calls to `getObjectById`, it never uses **multiple** mode, because the additional selects can be made lazily just as efficiently. This mode only increases efficiency over **single** mode when multiple objects are being loaded.

14.2.1. Configuring Eager Fetching

You can control Kodo's default eager fetch mode through the `kodo.EagerFetchMode` configuration property. Set this property to one of the mode names described in the previous section: `none`, `single`, `multiple`. If left unset, it defaults to `multiple` (assuming you have purchased the performance pack).

You can also override the default eager fetch mode 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: single
```

Example 14.3. Setting the Eager Fetch Mode at Runtime

```
import kodo.runtime.*;

...

KodoQuery kq = (KodoQuery) pm.newQuery (Person.class, "address.state == \"TX\"");
kq.getFetchConfiguration ().setEagerFetchMode
    (FetchConfiguration.EAGER_FETCH_SINGLE);
Collection results = (Collection) kq.execute ();
```

14.2.2. Eager Fetching Considerations

There are three important limitations of eager fetching that you should consider. First, when Kodo uses multiple selects to gather eager data under the `multiple` eager fetch mode, lazy result set support is turned off. Kodo will fully process each of the result sets immediately. Note that this applies only to result sets where secondary eager selects are made; if there are no to-many relations in the configured fetch groups that require separate selects, then lazy result set support will remain on, regardless of the eager fetch mode setting.

Second, eager fetching can be *less* efficient than standard fetching when circular relations are included in the configured fetch groups. For example, if type A has a relation to type B, and B has a relation back to A, and both relations are in the default fetch group or a configured fetch group, Kodo will be most efficient with eager fetching turned off.

Finally, when Kodo eager-fetches a 1-1 relation, any recursive eager fetching from that relation is in `single` mode. Kodo cannot eager-fetch a 1-1, and then recursively eager-fetch a to-many relation from that 1-1.

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” \[365\]](#)

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” \[366\]](#)

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.

When using a Tangosol Coherence cache plug-in, all remote updating of cache information is delegated to the Coherence cache. See the descriptions of the different remote commit providers in [Section 14.4.1, “Remote Commit Provider Configuration” \[?\]](#) for details on multi-JVM cache synchronization options.

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” \[366\]](#)

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. Removed objects are moved to a soft reference map, so they may stick around for a little while longer. Additionally, objects that are pinned into the cache are not counted when determining if the cache size exceeds the maximum.

```
kodo.DataCache: true(CacheSize=5000)
```

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>
```

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.properties JDO configuration file
...
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", true, false).
```

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.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getConfiguration ().getDataCacheManager ().getDataCache (false).
    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.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getConfiguration ().getDataCacheManager ().getDataCache (false).
    unpin (JDOHelper.getObjectId (o));
```

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

Example 14.9. Evicting an Object from the DataCache

```
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
kpm.getConfiguration ().getDataCacheManager ().getDataCache (true).
    remove (JDOHelper.getObjectId (o));
```

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:

Example 14.10. Setting the Size of the Query Cache

```
kodo.QueryCache: CacheSize=1000
```

To disable the query cache completely, set the `kodo.QueryCache` property to `false`:

Example 14.11. 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.

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 14.12. Notifying the Query Cache of Altered Classes

```
import kodo.datacache.*;
import kodo.runtime.*;

...
```

```
KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
QueryCache cache = kpm.getConfiguration ().getDataCacheManager ().
    getQueryCache ();
Collection changed = new ArrayList (2);
changed.add (A.class);
changed.add (B.class);
cache.notifyChangedClasses (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 14.13. Dropping or Pinning Query Results

```
import kodo.datacache.*;
import kodo.runtime.*;

...

KodoPersistenceManager kpm = (KodoPersistenceManager) pm;
QueryCache cache = kpm.getConfiguration ().getDataCacheManager ().
    getQueryCache ();

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

QueryKey key2 = new QueryKey (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 14.14. 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. Tangosol Integration

The Kodo JDO data cache can integrate with Tangosol's Coherence caching system. To use Tangosol integration, set the `kodo.DataCache` configuration property to `tangosol`, with the appropriate plugin properties for your Tangosol setup. For example:

Example 14.15. Distributed Tangosol Cache Configuration

```
kodo.DataCache: tangosol(TangosolCacheName=kodo, TangosolCacheType=distributed)
```

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. Valid values are either `distributed` or `replicated`. Defaults to `distributed`.

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.

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.CacheImpl`. 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” \[367\]](#)

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 specification.
- Invoking `PersistenceManager.refresh` or `PersistenceManager.evict` or related methods *does not* result in the corresponding data being dropped from the data cache. The data 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. If you really want to force data out of the cache, you should use the data cache APIs (see the JavaDoc for details), not the persistence manager cache control APIs.
- 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 ();
```

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.naming.TopicConnectionFactory` factory to use for finding topics. Defaults to `java:/ConnectionFactory`

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 jdosupport@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.

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* [324]** details these interfaces, including the `FetchConfiguration` in **Section 10.5, “Fetch Configuration” [329]**.

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 ();
```

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 re-

quires 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 always be set to project -->
    <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>
    </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 extensions", then you do not need to perform the manually file copying or any other additional steps.

Warning

The Kodo JBuilder OpenTool only works in JBuilder 7 and 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\slstudio` 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` and `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 `xerces.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-1.0.1.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 settings 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, 2.0, and 2.1 of the Eclipse IDE, and version 4 and 5 of WebSphere Studio Application Developer. While the plugin may work on other IDEs/ 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_2.0.0` directory from the `eclipse` directory in your Kodo installation to the `plugins` directory of the IDE base directory. For example, if you installed Kodo at `C:\Kodo` and Eclipse at `C:\Eclipse`, one would take `C:\Kodo\eclipse\kodo.eclipse_2.0.0` directory and place it such that the new directory structure would be:
`C:\Eclipse\plugins\kodo.eclipse_2.0.0`.

Copy all the jars from the `lib` directory of your Kodo installation into this new directory. In addition, copy the JDBC driver of your choice into this directory as well. *Keep the driver filename in mind as you will need it later.*

In the Kodo plugin directory, modify the marked portion of `plugin.xml` to point to your JDBC driver .jar file.

Start the IDE. You should see a Kodo menu item, and Kodo listed as an available view. If not, you may need to configure your perspective to include those items. To manually configure your perspective for WSAD 4, perform the following steps. These steps are likely to work for other versions of Eclipse, but have been written with WSAD 4 in mind.

1. Click on Perspective -> Customize... to open the Customize Perspective dialog box.
2. Open up the Views checkbox.
3. Select Kodo from the list of available views (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` file by selecting it 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 a `.jdo` file 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** 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 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

Optimize database indexes <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>
Use the best JDBC driver <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>
JVM optimizations <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 http://java.sun.com/docs/hotspot/PerformanceFAQ.html.</p>
Use the data cache <i>performance, scalability</i>	<p>Using Kodo's data caching and query caching 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>
Disable logging, performance tracking <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>
Set IgnoreCache to true, or set FlushBe-	<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>foreQueries to true</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>Configure <code>kodo.ConnectionRetainMode</code> appropriately</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>Ensure that batch updates are available</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>Use single-table inheritance</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. 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>High increment in the sequence factory</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>Use optimistic transactions</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>Use query aggregates and projections</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 ap-</p>

<i>performance, scalability</i>	plication.
Perform nontransactional data reads outside datastore (pessimistic) transactions <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.
Always close persistence managers, extent iterators, and query results <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>Example 16.1. Explicitly Closing Resources</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>
Optimize connection pool settings <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.
Utilize the persistence manager cache <i>performance, scalability</i>	When possible and appropriate, re-using persistence managers and setting the RetainValues configuration option to <code>true</code> may result in significant performance gains, since the persistence manager's built-in object cache will be used.
Enable multithreaded operation only when necessary <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.
Enable large data set handling <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 large result set handling where appropriate can dramatically speed up your application, since Kodo will bring the data sets into memory from the database only as necessary.
Disable large data set	If you have enabled scrollable result sets and on-demand loading but do you not require it, consider

handling <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>
Develop a custom class indicator, use the metadata-value indicator with short symbolic constants, or do not use class indicators <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 metadata-value 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>Example 16.2. Disabling the Class Indicator</p> <pre><jdo> <package name="com.xyz"> <class name="NoSubclasses"> <extension vendor-name="kodo" key="jdbc-class-ind-name" value="none"/> <!-- rest of class metadata --> </class> </package> </jdo></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>
Use the Dynamic-SchemaFactory <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 kodo.jdbc.SchemaFactory configuration property to dynamic 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.</p>
Do not use XA transactions <i>performance, scalability</i>	<p>XA transactions 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>
Use Sets instead of List/Collections <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 Set, SortedSet, HashSet, or TreeSet.</p>
Use JDOQL parameters instead of encoding search data in filter strings <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 PreparedStatements, 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> <p>Example 16.3. Appropriate use of JDOQL parameters</p>

```
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");
        query.declareParameters ("String fname, String 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 + \"");
        query.declareParameters ("String fname, String lname");
        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 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 398
 - 1.1. Using Application Identity 399
 - 1.2. Using JDO with Java Server Pages (jsp) 400
 - 1.3. Custom Proxies 401
 - 1.4. JDO Enterprise Java Beans Facade 402
 - 1.5. Customizing Logging 403
 - 1.6. Custom Sequence Factory 404
 - 1.7. Using Externalization to Persist Second Class Objects 405
 - 1.8. Using Persistent Classes Without Enhancement 406
 - 1.9. XDoclet Integration 407
 - 1.10. Custom Mappings 408
 - 1.11. Example of full-text searching in JDO 409
 - 1.12. Management / Monitoring 410

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. 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.

1.8. 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.

1.9. 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.

1.10. 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 special license. Contact sales@solarmetric.com for details. Custom field mappings do not require a special license.

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.

1.11. 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.

1.12. Management / Monitoring

This sample shows how to use the Kodo Monitoring technology preview.

Steps to use:

- 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.monitoring.SeedDatabase`
- Add the following line to your `kodo.properties` file: `kodo.ManagementUI: gui`
- `java samples.monitoring.MonitorSample`

In order to see an example of remote monitoring:

- 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.monitoring.SeedDatabase`
- Add the following line to your `kodo.properties` file: `kodo.ManagementServer: true`
- In one window, run: `java samples.monitoring.MonitorSampleRemote`
- Within a few seconds, in a second window, run: `remotemanagementtool`

Note that the Kodo PersistenceManagerFactory DataCache also implements the Watchable interface. You can see an example of DataCache monitoring 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 Watchable` can be found in `QueryThread.java`. A `TimeWatch Watchable` allows for monitoring named code blocks.

Part VII. Kodo Development Workbench Guide

Table of Contents

1. Introduction to the Kodo Development Workbench	cdxiv
1.1. Kodo Development Workbench Requirements	cdxv
1. Running and Configuring Kodo Development Workbench	416
1.1. Starting Kodo Development Workbench From the Command Line	417
1.2. Configuring Kodo Development Workbench	418
1.3. Standalone Configuration Tool	419
2. Getting Started with Kodo Development Workbench	420
2.1. Beginning The Kodo Development Workbench Tutorial	421
2.2. Getting Familiar with Kodo Development Workbench	422
2.3. The MetaData Explorer	423
2.4. The Schema Explorer	424
2.5. Kodo Development Workbench Logging	425
2.6. The Editor	426
2.7. Running The Tutorial	427
3. Root MetaData Actions	428
3.1. Mount JDO File	429
3.2. Unmount Files... ..	430
3.3. Create MetaData	431
3.4. Import Mapping Info	432
4. MetaData Actions	433
4.1. Enhance	434
4.2. Edit MetaData	435
4.3. Add - Recreate Mapping Info	436
4.4. Export Mapping Info	437
4.5. Drop Mapping Info	438
4.6. Remove MetaData	439
4.7. Build Schema From Mapping	440
4.8. Visualize Mapping	441
5. Root Schema Actions	442
5.1. Run SchemaTool	443
5.2. Refresh Schema From DB	444
5.3. Create DB Script	445
5.4. Create Change Script	446
5.5. Reverse Map Schema	447
5.6. Import .schema file	448
5.7. Export to .schema file	449
6. Schema Actions	450
6.1. Drop Schema Object	451
6.2. Edit Table	452
6.3. Add	453
7. The Editors	454
7.1. MetaData Editor	455
7.2. The Mapping Editor	456
7.3. The Table Editor	457

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.

Note

Note that this version of Kodo Workbench is a technology preview of a forthcoming Kodo product. This release may contain some usability and stability bugs. In addition, the tool may be missing some features that will be in the final release version. Please report all bugs and suggestions in the `solarmetric.kodo.beta` newsgroup located at news.solarmetric.com as Kodo Workbench will not be "officially" supported until a future date.

1.1. Kodo Development Workbench Requirements

This tutorial requires that JDK 1.3 or greater be installed on your computer, 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`.

A key concept for Kodo Workbench is that it uses the same environment as the one you started it from. For example, Kodo Workbench will not pick up classes that are compiled after Kodo Workbench has started. This gives you a very close approximation to how Kodo will behave at runtime.

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. If unspecified, this will default to base directory setting listed below. We highly recommend specifying an absolute path to a dedicated folder so that you do not lose your settings.
- `-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 the current working directory.
- `-UI/-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 dialog will appear to select, create, and edit a **configuration file** to configure Kodo. Enter a valid license key in the `General` pane, as well as connection information in the `Connecting` pane.

Select `Validate` to test your configuration and to enable the `Finish` button. Upon completion, Kodo Workbench will ask you to save your configuration to a file. Enter a file, and Kodo Workbench will remember that file location to avoid entering this wizard in the future.

You can also load configuration values from an existing `.properties` file by using the `Select` button. In addition to filling out the fields appropriately, this will also enable you to finish the configuration wizard.

If you want to export the configuration to another file that will *not* be used by Kodo Workbench, select `Write Copy`. And if you want to start with an empty configuration, select `New`.

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.

You can always return to this dialog to select a new configuration for future uses of Kodo Workbench selecting the `Configuration Tool` menu item from the `Configuration` menu.

1.3. Standalone Configuration Tool

Kodo includes a standalone version of the configuration tool. This tool is by executing the following command from the prompt:

```
configurationtool
```

The configuration tool takes in the standard command-line arguments of the configuration framework (see [Section 2.3, “Command Line Configuration” \[172\]](#)).

This tool allows you to load, edit, test, and save configurations to `.properties` files.

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 that we have generated our classes, 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 /path/to/kodo.properties
```

where `/path/to/` is substituted with the path to your `kodo.properties` file. 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 the majority of the screen should be taken up by the `Editor` pane. Each pane can be resized, expanded, and collapsed within the main window. Most of your work will happen within these three 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

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.

Individual class nodes from the explorer can be dragged and dropped into class text fields, such as assigning a persistence capable superclass.

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. Press the Next button to bring the final screen where you can tailor the metadata before it gets written to file. Select Finish for now.

You have created metadata for the `Passenger` class. You should see `Passenger` listed in Kodo Workbench's explorer represented by a blue Kodo *K* under a yellow label of the package name. This *K* will be the icon for JDO metadata throughout the system.

This metadata file is now persistently part of Kodo Workbench. If one 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. There are a number of *MetaData Actions* to try and use, all of which will be detailed further in a later section.

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`.

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'll be specifying the element type for this `java.util.Set` field. Enter `java.lang.String` in the corresponding 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.

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 **Add - Recreate Mapping Info**. 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 Workbench 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 `Add - Recreate Mapping Info`, 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 one 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. One can edit tables, add new schema objects, and more.

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 `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 `org.apache.commons.logging.LogFactory` (see the **Logging** section for more details) that will log all messages and then echo the messages appropriately to your current Jakarta Commons 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.

Each tab corresponds to a different Log instance / scope. Each tabbed panel provides a context menu by right clicking on the respective tab. This menu allows you to reset, close, and adjust verbosity (the selected verbosity is in red).

Note

The verbosity of each Log panel is independent of your own Logging configuration. So you may be logging at the TRACE to file or the console, but at ERROR in Kodo Workbench.

2.6. The Editor

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 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. The **Visualize** section of the `MetaData Actions` section will contain more details on how the Visualizer works.

2.7. Running The Tutorial

Before running the tutorial test program, we'll first synchronize Kodo Workbench's schema with your database schema. Switch to the *Schema Explorer*. Right click on Schemas and select `Run SchemaTool`. Select `Run` and then select `Ok` from the confirmation screen. You can see the SQL statements executed in the SQL tab of the Logging pane.

Now that the runtime schema is up to date, 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 SchemaFactory. For example in a database-based SchemaFactory, 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 metadata and corresponding **mapping** that Kodo will use at runtime to persist your data.












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. A basic amount of information is shown for each, such as column types and field names.

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 visualization toggles will appear. Select what portions of the mappings and their related objects for the system to generate. You can also view a legend of line colors 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.

In addition to the table and class nodes, one may also see yellow mapping nodes. For complex field mappings, Kodo Workbench will generate this node indicating the type of mapping used for the linked field. These lines are in green.

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
 - Orange	Reference Source
 - Red	Reference Target
 - Magenta	Persistent related field type
 - Yellow	Element persistent type
 - Dark Purple	Key persistent type

This tool gives you a way to visualize the data in the way in which Kodo will see it at runtime. This can help prevent confusion as to what columns and tables Kodo expects from your schema. In addition, one can see how your classes will interact with your database and how cross-dependencies will line up.

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 one 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, irregardless 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.
- *Option 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 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**

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
Borland JDataStore	6.0	Borland JDataStore	6.0
DB2	8.1	IBM DB2 JDBC Universal Driver	1.0.581
Hypersonic Database Engine	1.7.0	Hypersonic	1.7.0
Informix Dynamic Server	9.30.UC10	Informix JDBC driver	2.21.JC2
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	2.0 (2.0.14)
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	4.2 (4.2)

B.1. JDataStore

Example B.1. 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.2. IBM DB2

Example B.2. Example properties for IBM DB2

```
javax.jdo.option.ConnectionDriverName: com.ibm.db2.jcc.DB2Driver  
javax.jdo.option.ConnectionURL: jdbc:db2://SERVER_NAME:SERVER_PORT/DB_NAME
```

B.2.1. 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.3. Hypersonic

Example B.3. Example properties for Hypersonic

```
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionURL: jdbc:hsqldb:DB_NAME
```

B.3.1. Known issues with Hypersonic

- Hypersonic does not properly support foreign key constraints.
- Hypersonic does not support pessimistic locking, so non-optimisitic transactions will fail unless the `SimulateLocking` property is set for the **`kodo.jdbc.DBDictionary`**

B.4. Informix

Example B.4. 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.4.1. Known issues with Informix

-

B.5. Microsoft Access

Example B.5. 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.5.1. Known issues with Microsoft Access

- Using the Sun JDBC-ODBC bridge to connect is not supported.

B.6. Microsoft SQL Server

Example B.6. 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
```

B.6.1. 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.
- 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.7. Microsoft FoxPro

Example B.7. Example properties for Microsoft FoxPro

```
javax.jdo.option.ConnectionDriverName: com.ddtek.jdbc.sequelink.SequeLinkDriver
javax.jdo.option.ConnectionURL: jdbc:sequelink://SERVER_NAME:SERVER_PORT
```

B.7.1. Known issues with Microsoft FoxPro

- Using the Sun JDBC-ODBC bridge to connect is not supported.

B.8. MySQL

Example B.8. Example properties for MySQL

```
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL: jdbc:mysql://SERVER_NAME/DB_NAME
```

B.8.1. 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.
- Using `isEmpty()` in queries made against an extent will fail because SQL sub-selects are not supported.
- Rollback due to database error or optimistic lock violation is not supported unless the table type is one of the MySQL transactional types. Explicit calls to `rollback()` before a transaction has been committed, however, are always supported.
- Floats and doubles may lose precision when stored in some data stores.
- When storing a field of type `java.math.BigDecimal`, some data stores will add extraneous trailing 0 characters, causing an equality mismatch between the field that is stored and the field that is retrieved.

B.9. Oracle

Example B.9. 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.9.1. 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.10. Pointbase

Example B.10. 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.10.1. Known issues with Pointbase

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

B.11. PostgreSQL

Example B.11. Example properties for PostgreSQL

```
javax.jdo.option.ConnectionDriverName: org.postgresql.Driver  
javax.jdo.option.ConnectionURL: jdbc:postgresql://SERVER_NAME:5432/DB_NAME
```

B.11.1. 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.12. Sybase Adaptive Server

Example B.12. 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.12.1. 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.

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	SQL0803N One or more values in the INSERT statement, UPDATE statement, or foreign key update caused by a DELETE statement are not valid because the primary key, unique constraint or unique index identified by "1" constrains table "%s" from having duplicate rows for those columns.	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
DB2	-511	42829	SQL0511N The FOR UPDATE clause is not allowed because the table specified by the cursor cannot be modified.	A datastore transaction read was attempted on a table that is marked as read-only. Either read the data outside of a transaction, or use optimistic transactions.
DB2	-401	42818	SQL0401N The data types of the operands for the operation ">=" are not compatible.	A mathematical comparison query was attempted on a field whose mapping was to a non-numeric field, such as VARCHAR. DB2 disallows such queries.
DB2	-302	22003	SQL0302N The value of a host variable in the EXECUTE or OPEN statement is too large for its corresponding use.	Possible attempt to store a string of a length greater than is allowed by the database's column definition. If creation is done via the mapping-tool, ensure that the jdbc-size JDO metadata extension specifies a large enough size for the column.
DB2	-204	42S02	SQL0204N "%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 mapping 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 SQL statement ordering .
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	1196	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 DBDictionary 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 MySQL deadlock documentation.
MySQL	0	08S01	Communication link failure: java.io.IOException	The TCP connection underlying the JDBC Connection has been closed, possibly due to a timeout. If using Kodo's default data source, connection testing can be configured via the ConnectionFactoryProperties 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 kodo.jdbc.DataSourceMode 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 mapping documentation.
Oracle	1722	42000	ORA-01722: invalid number	A number that Oracle cannot store has been persisted. This can happen when a String field in the persistent class is mapped to an Oracle column of type NUMBER and the String value is not numeric.
Oracle	1000	72000	ORA-01000: maximum open cursors exceeded	<p>Oracle limits the number of statements that can be open at any given time, and the application has made requests that keep open more statements than Oracle can handle. This can be resolved in one of the following ways:</p> <ol style="list-style-type: none"> 1. Increase the number of cursors allowed in the data-

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"> 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. 3. Decrease the value of the <code>MaxStatementCache</code> parameter in the ConnectionFactoryProperties configuration property.
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> fields have a limitation

Database	Error Code	SQL State	Message	Solution
				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 column.

Database	Error Code	SQL State	Message	Solution
SQLServer	0	08007	Can't start a cloned connection while in manual transaction mode.	Append ";SelectMethod=cursor" to the ConnectionURL. See the description of the problem on the Microsoft support site .
SQLServer			sp_cursorclose: The cursor identifier value provided (abcdef0) is not valid.	This can sometimes show up as a warning when Kodo is closing a 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.
SQLServer	0	HY000	Object has been	The TCP connection un-

Database	Error Code	SQL State	Message	Solution
			closed.	derlying 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 ConnectionFactoryProperties configuration property.
Sybase	311	ZZZZZ	The optimizer could not find a unique index which it could use to scan table '%s' for cursor 'jconnect_implicit_%d'	A pessimistic lock was attempted on a table that does not have a primary key (or other unique index). By default, the Kodo 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	2762		The 'CREATE TABLE' command is not allowed within a multi-statement transaction in the 'tempdb' database.	This may happen when running the schematool against a Sybase database that is not configured to allow schema-altering commands to be executed from within a transaction. This can be enabled by entering the command sp_dboption database_name,"ddl in tran", true from isql. See the Sybase documentation for allowing data definition commands in transactions .
Sybase	0	JZ0BE	JZ0BE: BatchUpdateException: Error 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 store the specified value

Database	Error Code	SQL State	Message	Solution
				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	2601	23000	Attempt to insert duplicate key row in object '%s' with unique index '%s'	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.

Appendix D. 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. 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. 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.1.2. 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” [373]**. 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 6.2.1.2, “dependent” [238]** 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.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.

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.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>
```

D.4. Storing Object-Relational Mapping Data

Please read the reference guide's description of the available mapping factories in **Section 7.2, “Mapping Factory”** [251]. 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.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.

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. 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.
- `jakarta-commons-collections-2.1.jar`: Collection classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/collections.html>.
- `jakarta-commons-lang-1.0.1.jar`: Utility classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/lang.html>.
- `jakarta-commons-logging-1.0.3.jar`: Logging wrapper classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/logging.html>.
- `jakarta-commons-pool-1.0.1.jar`: Pooling classes. Required for all development and runtime use of Kodo. See <http://jakarta.apache.org/commons/pool.html>.
- `jakarta-regexp-1.1.jar`: Regular expression library. Required for all development and runtime use of Kodo queries. See <http://jakarta.apache.org/regexp/>.
- `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” \[316\]](#) for more details. Distributed from <http://jline.sourceforge.net>.
- `jdo-1.0.1.jar`: Interfaces defined by the Java Data Objects standard. Required for all development and runtime use.
- `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/>.
- `log4j-1.2.6.jar`: Fast, configurable logging implementation. This library is not required by Kodo JDO. See <http://jakarta.apache.org/log4j/docs/>.
- `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” \[316\]](#).
- `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 G. Release Notes

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 7.4.2, “Non-Standard Joins” \[256\]](#)) 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 11.3, “Detach and Attach Callbacks” \[344\]](#).
 - Added support for simulating auto-increment fields under Oracle by using triggers. See [Section 4.3.2, “OracleDictionary parameters” \[207\]](#).

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” \[203\]](#).
 - 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, configuration 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” [316]**, 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” [378]** for details.
 - Introduced a cache for shareable query-related information. This improves query compilation times in many situations. See **Section 2.6.28, “kodo.QueryCompilationCache” [185]** 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” [359]** 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 13.3, “Direct SQL Execution” \[353\]](#) for details.
 - Technology preview of Kodo Management / Monitoring capability. See [Chapter 12, Management and Monitoring \[345\]](#) 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 13.4, “MethodQL” \[354\]](#) 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” [364]** 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 the **Query Aggregates and Projections** documentation for more 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 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 `PreparedStatement`s and batch updates whenever possible for very significant performance benefits. See the documentation for the `com.solarmetric.kodo.impl.jdbc.UsePreparedStatement`s and `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements` properties.
- Inverse one-to-one mappings are now supported. The field can now reside on either table corresponding to the related objects. If both sides of an one-to-one are marked as having an inverse, one field should be designated as read-only to indicate to the system the owning class and table for the given relational field.
- Logging is now done through the Apache commons project, which offers the ability to use an underlying logging mechanism, such as Apache Log4J, JDK 1.4's native logging, or simple file/stdout logging. It is now necessary to include the new `jakarta-commons-logging-1.0.2.jar` in the CLASSPATH. See the **Logging** chapter.
- Added a pluggable `SQLExecutionManager` architecture, which allows the developer to override the mechanism by which SQL is issued to the database. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass` property.
- There is now an option to automatically refresh the database schema during runtime, allowing the developer to skip the `schematool` step. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` property.
- Properties may be specified for a Driver with the `com.solarmetric.kodo.ConnectionProperties` property.
- A `javax.sql.DataSource` may be specified in the **`javax.jdo.option.ConnectionDriverName`** property, which will be customizable with bean-like entries in the `com.solarmetric.kodo.ConnectionProperties` property.
- The default transaction isolation for a JDBC connection can be overridden with the **`com.solarmetric.kodo.impl.jdbc.TransactionIsolation`** property.
- Kodo JDO now distributes a single jar for both the enterprise and standard edition, as well as datacache and query extensions.
- The `rd-metadatatool` can now be used to generate default JDO metadata for classes.
- The **`rd-schemagen`** tool used for reverse mapping classes from a schema has now been tested with the following databases: Hypersonic SQL 7.1, SQLServer (MS Beta 2 JDBC driver), Sybase, Oracle (9.0.1 JDBC driver), DB2, Postgres (7.3 Beta 3 JDBC driver).

Bugfixes

- `default-fetch-group="false"` is now respected for fields that default to `default-fetch-group="true"`.
- Traversing orphaned relations in data cache now behaves in the same way as traversing orphaned db relations -- the invalid relation is set to null.
- Changing a field to the same value as it was originally set to no longer constitutes dirtying that field. This means that subsequent flushes to the database will not necessarily re-write the same data.

- Class loading is performed in accordance with section 12.5 of the JDO specification.

Notable changes

- The API for implementing a `SequenceFactory` has changed. See the API documentation for [`com.solarmetric.kodo.impl.jdbc.SequenceFactory`](#).
- Persistent types are no longer enumerated, either in the data store or in a property. Classes are now dynamically added upon class initialization, via the `JDOImplHelper` class registration process. The `-register` and `-unregister` options to `schematool` are no longer needed. The `JDO_SCHEMA_METADATA` table is no longer used and can be dropped.

2.3.4

- New features
 - The R&D schema generator can now accept a list of tables to generate.
 - The R&D reverse mapping tool has additional options for using foreign key names to generate relation names, generating primitive wrapper-type fields if a column is nullable, and allowing primary keys on many-to-many join tables.
- Bugfixes
 - Fixed problems with many-to-many relations between tables that use vertical inheritance.
 - Fixed bug in `schematool` that caused it to not generate primary key columns in subclass tables when using `datastore identity + custom names + vertical inheritance`.
 - Fixed `serp` library conflict between reverse mapping tool and main Kodo libraries.
 - Fixed a reverse mapping tool bug in which column names that conflicted with Java keywords would result in the generation of uncompileable Java classes.
 - Fixed problem that caused read-only flag to be ignored in many-to-many relations.
 - Multi-table inheritance deletes are now performed from the leaf table in the inheritance chain up to the base table. Inserts are performed from the base table down to the leaf. This supports the common referential integrity model of establishing a foreign key relation from inherited tables to their parent tables.

2.3.3

- New features
 - The R&D schema generator can now accept a list of tables to generate.
- Bugfixes

- Fixed `null-value="default"` behavior.
- Fixed bugs that prevented removal of map elements through the key set, entry set, and values collection.
- Added more validation on static/final fields to metadata.
- Fixed memory leak in `serp` regarding soft and weak collections backed by `HashSets`.
- Multi-variable query issues resolved.
- Fixed bug that could cause optimistic lock version numbers to be incremented before successful transaction commit.
- The R&D reverse mapping tool now handles Oracle DATE columns correctly.

2.3.2

- New features
 - The new `com.solarmetric.kodo.CacheReferenceSize` property dictates a number of hard references to cached objects that the `PersistenceManager` will retain, in addition to its soft cache.
 - Added 'all' option to unregister action of `schematool` Ant task. This option allows all classes in the current persistent types list to be unregistered, regardless of whether or not those classes are currently in the classpath.
 - Added `com.solarmetric.kodo.UseSoftTransactionCache` to configure whether or not Kodo should maintain soft references to transactional items that have not been dirtied. This now defaults to false; previous versions of Kodo always used soft references for non-dirty transactional items.
- Bugfixes
 - The `jdodoclet` task no longer creates JDO metadata entries for final or static fields, or for transient fields that do not have a `jdo:persistence-modifier` tag.
 - The `jdoc` task no longer attempts to enhance classes that have already been enhanced.
 - Several default property values were being set improperly.

2.3.1

- New features
 - Smart proxies for set and map fields. Smart proxies better optimize database updates when persistent set and map fields are modified.
 - TCP, JMS-based distributed `DataCache` implementations.
 - All `DataCache` implementations now use an LRU cache with a configurable maximum size.

- Customizable tracked instance proxies.
- Alpha release of upcoming reverse mapping tool for creating persistent class definitions, metadata, and mapping extensions from an existing schema.
- Cache object `com.solarmetric.kodo.runtime.datacache.PMFactoryCache` is now named `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`.
- Bugfixes
 - A Query with an unspecified filter defaults to a filter of "true", rather than "false".
 - A Query with an unspecified candidate Extent but a specified candidate Class will automatically create an Extent of the appropriate type with subclasses turned on.
 - Various bugs related to compiled queries with null arguments or no parameters have been resolved.
 - Various InstanceCallbacks interface bugs have been resolved.
 - Ant schematool and jdoc tasks deal with the Ant ClassLoader system better.
- Notable changes
 - Made `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 ex-

tensions 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 H. 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.