

Java™ Data Objects

JSR 12

Version 1.0.1

Java Data Objects Expert Group

Specification Lead: **Craig Russell,**
Sun Microsystems Inc.

Technical comments:
jdo-comments@sun.com
Process comments:
community-process@sun.com



Sun Microsystems, Inc.
4140 Network Circle
Santa Clara, California 95054
408 276-5638 fax: 408 276-7191

Java™ Data Objects Specification ("Specification")

Version: 1.0.1

Status: FCS

Release: May 31, 2003

Copyright 2000-2003 Sun Microsystems, Inc.
4140 Network Circle, Santa Clara, California 95404, U.S.A.
All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control Guidelines as set forth in the Terms of Use on Sun's website. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's intellectual property rights that are essential to practice the Specification, to internally practice the Specification for the purpose of designing and developing your Java applets and applications intended to run on the Java platform or creating a clean room implementation of the Specification that:

(i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages or other packages defined by the Specification; (vi) satisfies all testing requirements available from Sun relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. The Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Acknowledgments

I have come to know Rick Cattell during many shared experiences in the Java database standards arena. Rick is a Distinguished Engineer at Sun Microsystems and has been the database guru and Enterprise Cardinal in the Java “Church” for many years. I am deeply in his debt for his many contributions to JDO, both technical and organizational.

I want to thank the experts on the JDO expert group who contributed ideas, APIs, feedback, and other valuable input to the standard, especially Heiko Bobzin, Constantine Plotnikov, Luca Garulli, Philip Conroy, Steve Johnson, Michael Birk, Michael Rowley, Gordan Vosicki, and Martin McClure.

I want to recognize Michael Bouschen, David Jordan, David Ezzio, Dave Bristor, and Jeff Norton for their careful review of JDO for consistency, readability, and usability. Without their contributions, JDO would not have been possible.

Table of Contents

1 Introduction	15
1.1 Overview	15
1.2 Scope	16
1.3 Target Audience	16
1.4 Organization	16
1.5 Document Convention	16
1.6 Terminology Convention	17
2 Overview	18
2.1 Definitions	18
2.1.1 JDO common interfaces.	18
2.1.2 JDO in a managed environment.	19
Enterprise Information System (EIS)	19
EIS Resource.	19
Resource Manager (RM).	19
Connection	20
Application Component	20
Session Beans	20
Entity Beans	20
Helper objects	20
Container.	20
2.2 Rationale	20
2.3 Goals	21
3 JDO Architecture	23
3.1 Overview	23
3.2 JDO Architecture	24
3.2.1 Two tier usage	24
3.2.2 Application server usage	24
Resource Adapter	24
Pooling	25
Contracts	25
4 Roles and Scenarios	28
4.1 Roles	28
4.1.1 Application Developer.	28
4.1.2 Application Component Provider	28
4.1.3 Application Assembler.	28
4.1.4 Deployer.	29
4.1.5 System Administrator	29
4.1.6 JDO Vendor	29
4.1.7 Connector Provider	29
4.1.8 Application Server Vendor	29
4.1.9 Container Provider.	30
4.2 Scenario: Embedded calendar management system	30
4.3 Scenario: Enterprise Calendar Manager	31
5 Life Cycle of JDO Instances	33
5.1 Overview	33

Table of Contents

5.2 Goals	34
5.3 Architecture:	34
JDO Instances	34
JDO State Manager	34
JDO Managed Fields	35
5.4 JDO Identity	35
Three Types of JDO identity	35
Uniquing	36
Change of identity	37
JDO Identity Support	37
5.4.1 Application (primary key) identity	37
5.4.2 Datastore identity	38
5.4.3 Nondurable JDO identity	39
5.5 Life Cycle States	40
Datastore Transactions	40
5.5.1 Transient (Required)	40
5.5.2 Persistent-new (Required)	40
5.5.3 Persistent-dirty (Required)	41
5.5.4 Hollow (Required)	42
5.5.5 Persistent-clean (Required)	42
5.5.6 Persistent-deleted (Required)	42
5.5.7 Persistent-new-deleted (Required)	43
5.6 Nontransactional (Optional)	43
5.6.1 Persistent-nontransactional (Optional)	44
5.7 Transient Transactional (Optional)	45
5.7.1 Transient-clean (Optional)	45
5.7.2 Transient-dirty (Optional)	46
5.8 Optimistic Transactions (Optional)	46
6 The Persistent Object Model	54
6.1 Overview	54
6.2 Goals	55
6.3 Architecture	55
PersistenceCapable interface	55
First Class Objects and Second Class Objects	56
First Class Objects	56
Second Class Objects	56
Arrays	57
Primitives	58
Interfaces	58
6.4 Field types of persistence-capable classes	58
6.4.1 Nontransactional non-persistent fields	58
6.4.2 Transactional non-persistent fields	58
6.4.3 Persistent fields	58
Primitive types	58
Immutable Object Class types	59
Mutable Object Class types	59

Table of Contents

PersistenceCapable Class types	59
Object Class type	59
Collection Interface types	59
Other Interface types	60
Arrays	60
6.5 Inheritance	60
7 PersistenceCapable	62
7.1 Persistence Manager	62
7.2 Make Dirty	62
7.3 JDO Identity	63
7.4 Status interrogation	63
7.4.1 Dirty	63
7.4.2 Transactional	63
7.4.3 Persistent	63
7.4.4 New	64
7.4.5 Deleted	64
7.5 New instance	64
7.6 State Manager	65
7.7 Replace Flags	65
7.8 Replace Fields	65
7.9 Provide Fields	65
7.10 Copy Fields	65
7.11 Static Fields	65
7.12 JDO identity handling	66
interface ObjectIdFieldSupplier	66
interface ObjectIdFieldConsumer	67
interface ObjectIdFieldManager extends ObjectIdFieldSupplier, ObjectIdFieldConsumer	67
8 JDOHelper	68
8.1 Persistence Manager	68
8.2 Make Dirty	68
8.3 JDO Identity	68
8.4 Status interrogation	69
8.4.1 Dirty	69
8.4.2 Transactional	69
8.4.3 Persistent	69
8.4.4 New	69
8.4.5 Deleted	69
8.5 PersistenceManagerFactory methods	69
9 JDOImplHelper	72
9.1 JDOImplHelper access	72
9.2 Metadata access	72
9.3 Persistence-capable instance factory	73
9.4 Registration of PersistenceCapable classes	73
9.4.1 Notification of PersistenceCapable class registrations	73

Table of Contents

RegisterClassEvent	74
RegisterClassListener	74
9.5 Security administration	74
9.6 Application identity handling	75
10 InstanceCallbacks	76
10.1 jdoPostLoad	76
10.2 jdoPreStore	76
10.3 jdoPreClear	76
10.4 jdoPreDelete	77
11 PersistenceManagerFactory	78
11.1 Interface PersistenceManagerFactory	78
Construction by Properties	79
11.2 ConnectionFactory	80
11.3 PersistenceManager access	81
11.4 Close the PersistenceManagerFactory	81
11.5 Non-configurable Properties	82
11.6 Optional Feature Support	82
11.7 Static Properties constructor	83
12 PersistenceManager	85
12.1 Overview	85
12.2 Goals	85
12.3 Architecture: JDO PersistenceManager	85
12.4 Threading	86
12.5 Class Loaders	86
12.6 Interface PersistenceManager	87
Null management	87
12.6.1 Cache management	88
12.6.2 Transaction factory interface	89
12.6.3 Query factory interface	89
12.6.4 Extent Management	90
12.6.5 JDO Identity management	90
12.6.6 JDO Instance life cycle management	92
Make instances persistent	92
Delete persistent instances	93
Make instances transient	93
Make instances transactional	93
Make instances nontransactional	94
12.7 Transaction completion	94
12.8 Multithreaded Synchronization	94
12.9 User associated object	95
12.10 PersistenceManagerFactory	95
12.11 ObjectId class management	95
13 Transactions and Connections	96
13.1 Overview	96
13.2 Goals	96

Table of Contents

13.3 Architecture: PersistenceManager, Transactions, and Connections	96
Connection Management Scenarios	97
Native Connection Management.	97
Non-native Connection Management.	98
Optimistic Transactions	98
13.4 Interface Transaction	99
13.4.1 PersistenceManager	99
13.4.2 Transaction options	99
Nontransactional access to persistent values	99
Optimistic concurrency control.	99
Retain values at transaction commit.	100
Restore values at transaction rollback.	100
13.4.3 Synchronization	100
13.4.4 Transaction demarcation	101
Non-managed environment.	101
Managed environment	102
13.5 Optimistic transaction management	102
14 Query	104
14.1 Overview	104
14.2 Goals	104
14.3 Architecture: Query	105
14.4 Namespaces in queries	106
14.5 Query Factory in PersistenceManager interface	106
14.6 Query Interface	107
Persistence Manager	107
Query element binding	107
Query options	108
Query compilation	109
14.6.1 Query execution.	109
14.6.2 Filter specification	110
14.6.3 Parameter declaration.	113
14.6.4 Import statements.	113
14.6.5 Variable declaration.	113
14.6.6 Ordering statement.	114
14.6.7 Closing Query results.	114
14.7 Examples:	115
14.7.1 Basic query.	115
14.7.2 Basic query with ordering.	115
14.7.3 Parameter passing.	115
14.7.4 Navigation through single-valued field.	116
14.7.5 Navigation through multi-valued field.	116
14.7.6 Membership in a collection	116
15 Extent	118
15.1 Overview	118
15.2 Goals	118
15.3 Interface Extent	119

Table of Contents

16 Enterprise Java Beans	120
16.1 Session Beans	120
16.1.1 Stateless Session Bean with Container Managed Transactions	121
16.1.2 Stateful Session Bean with Container Managed Transactions	121
16.1.3 Stateless Session Bean with Bean Managed Transactions	121
16.1.4 Stateful Session Bean with Bean Managed Transactions	122
16.2 Entity Beans	122
16.2.1 BMP Entity Bean life cycle	122
17 JDO Exceptions	125
17.1 JDOException	125
17.1.1 JDOFatalException	126
17.1.2 JDOCanRetryException	126
17.1.3 JDOUnsupportedOptionException	126
17.1.4 JDOUserException	126
17.1.5 JDOFatalUserException	127
17.1.6 JDOFatalInternalException	127
17.1.7 JDODataStoreException	127
17.1.8 JDOFatalDataStoreException	127
17.1.9 JDOObjectNotFoundException	127
17.1.10 JDOOptimisticVerificationException	127
18 XML Metadata	128
18.1 ELEMENT jdo	129
18.2 ELEMENT package	129
18.3 ELEMENT class	129
18.4 ELEMENT field	130
Default persistence-modifier	130
18.4.1 ELEMENT collection	131
18.4.2 ELEMENT map	132
18.4.3 ELEMENT array	132
18.5 ELEMENT extension	132
18.6 The Document Type Descriptor	133
18.7 Example XML file	134
19 Portability Guidelines	135
19.1 Optional Features	135
19.1.1 Optimistic Transactions	135
19.1.2 Nontransactional Read	135
19.1.3 Nontransactional Write	135
19.1.4 Transient Transactional	135
19.1.5 RetainValues	135
19.1.6 IgnoreCache	135
19.2 Object Model	135
19.3 JDO Identity	136
19.4 PersistenceManager	136
19.5 Query	136
19.6 XML metadata	137

Table of Contents

19.7 Life cycle	137
19.8 JDOHelper	137
19.9 Transaction	137
20 JDO Reference Enhancer	138
20.1 Overview	138
20.2 Goals	138
20.3 Enhancement: Architecture	139
20.4 Inheritance	142
20.5 Field Numbering	142
20.6 Serialization	142
20.7 Cloning	143
20.8 Introspection (Java core reflection)	144
20.9 Field Modifiers	144
20.9.1 Non-persistent	144
20.9.2 Transactional non-persistent	144
20.9.3 Persistent	144
20.9.4 PrimaryKey	145
20.9.5 Embedded	145
20.9.6 Null-value	145
20.10 Treatment of standard Java field modifiers	146
20.10.1 Static	146
20.10.2 Final	146
20.10.3 Private	146
20.10.4 Public, Protected	146
20.11 Fetch Groups	146
20.12 jdoFlags Definition	147
20.13 Exceptions	147
20.14 Modified field access	147
20.15 Generated fields in least-derived PersistenceCapable class	148
20.16 Generated fields in all PersistenceCapable classes	148
Generated static initializer	149
20.17 Generated methods in least-derived PersistenceCapable class	149
20.18 Generated methods in PersistenceCapable root classes and all classes that declare objectid-class in xml metadata:	150
20.19 Generated methods in all PersistenceCapable classes	151
20.20 Example class: Employee	154
20.20.1 Generated fields	154
20.20.2 Generated static initializer	155
20.20.3 Generated interrogatives	155
20.20.4 Generated jdoReplaceStateManager	156
20.20.5 Generated jdoReplaceFlags	157
20.20.6 Generated jdoNewInstance helpers	157
20.20.7 Generated jdoGetManagedFieldCount	157
20.20.8 Generated jdoGetXXX methods (one per persistent field)	158
20.20.9 Generated jdoSetXXX methods (one per persistent field)	159
20.20.10 Generated jdoReplaceField and jdoReplaceFields	160

Table of Contents

20.20.11	Generated jdoProvideField and jdoProvideFields	161
20.20.12	Generated jdoCopyField and jdoCopyFields methods	162
20.20.13	Generated writeObject method	163
20.20.14	Generated jdoPreSerialize method.	163
20.20.15	Generated jdoNewObjectIdInstance	163
20.20.16	Generated jdoCopyKeyFieldsToObjectId	164
20.20.17	Generated jdoCopyKeyFieldsFromObjectId	164
21	Interface StateManager	165
21.1	Overview	165
21.2	Goals	165
	Clone support	165
21.3	StateManager Management	165
21.4	PersistenceManager Management	166
21.5	Dirty management	166
21.6	State queries	166
21.7	JDO Identity	167
21.8	Serialization support	167
21.9	Field Management	167
21.9.1	User-requested value of a field	168
21.9.2	User-requested modification of a field	168
21.9.3	StateManager-requested value of a field	169
21.9.4	StateManager-requested modification of a field	170
22	JDOPermission	171
23	JDO Query BNF	172
23.1	Grammar Notation	172
23.2	Parameter Declaration	172
23.3	Variable Declaration	173
23.4	Import Declaration	173
23.5	Ordering Specification	173
23.6	Filter Expression	174
23.7	Types	177
23.8	Literals	177
23.9	Names	178
24	Items deferred to the next release	179
24.1	Nested Transactions	179
24.2	Savepoint, Undosavepoint	179
24.3	Inter-PersistenceManager References	179
24.4	Enhancer Invocation API	179
24.5	Prefetch API	179
24.6	BLOB/CLOB datatype support	179
24.7	Managed (inverse) relationship support	179
24.8	Case-Insensitive Query	180
24.9	String conversion in Query	180
24.10	Read-only fields	180
24.11	Enumeration pattern	180

Table of Contents

24.12 Non-static inner classes	180
24.13 Projections in query	181
24.14 LogWriter support	181
24.15 New Exceptions	181
24.16 Distributed object support	181
24.17 Object-Relational Mapping	181
Appendix A: References	182
Appendix B: Design Decisions	183
B.1 Enhancer	183
B.1 PersistenceCapable	183
B.1 Collection Factory	184
Appendix C: Revision History	185
C.1 Changes since Draft 0.1	185
C.1 Changes since Draft 0.2	185
C.1 Changes since Draft 0.3	185
C.1 Changes since Draft 0.4	185
C.1 Changes since Draft 0.5	186
C.1 Changes since Draft 0.6 (Participant Review Draft)	187
C.1 Changes since Draft 0.7	187
C.1 Changes since Draft 0.8	188
C.1 Changes since Draft 0.9	188
C.1 Changes since draft 0.91	189
C.1 Changes since draft 0.92	190
C.1 Changes since draft 0.93	190
C.1 Changes since draft 0.94	191
C.1 Changes since draft 0.95 (Proposed Final Draft)	192
C.1 Changes since draft 0.96	192
C.1 Changes since draft 0.97	193
C.1 Changes since Approved Draft	194

List of Figures

Figure 25: Standard plug-and-play between application programs and EISes using JDO	21
Figure 26: Overview of non-managed JDO architecture	23
Figure 27: Contracts between application server and native JDO resource adapter.	26
Figure 28: Contracts between application server and layered JDO implementation	26
Figure 29: Scenario: Embedded calendar manager	30
Figure 30: Scenario: Enterprise Calendar Manager	32
Figure 31: Life Cycle: New Persistent Instances	49
Figure 32: Life Cycle: Transactional Access	50
Figure 33: Life Cycle: Datastore Transactions	50
Figure 34: Life Cycle: Optimistic Transactions	50
Figure 35: Life Cycle: Access Outside Transactions	51
Figure 36: Life Cycle: Transient Transactional	51
Figure 37: JDO Instance State Transitions	52
Figure 38: Instantiated persistent objects	54
Figure 39: Transactions and Connections.	98
Figure 40: Enterprise Java Beans: Entity Bean relationships	124

1 Introduction

Java is a language that defines a runtime environment in which user-defined classes execute. The instances of these user-defined classes might represent real world data. The data might be stored in databases, file systems, or mainframe transaction processing systems. These data sources are collectively referred to as Enterprise Information Systems (EIS). Additionally, small footprint environments often require a way to manage persistent data in local storage.

The data access techniques are different for each type of data source, and accessing the data presents a challenge to application developers, who currently need to use a different Application Programming Interface (API) for each type of data source.

This means that application developers need to learn at least two different languages to develop business logic for these data sources: the Java programming language; and the specialized data access language required by the data source.

Currently, there are two Java standards for storing Java data persistently: serialization and JDBC. Serialization preserves relationships among a graph of Java objects, but does not support sharing among multiple users. JDBC requires the user to explicitly manage the values of fields and map them into relational database tables.

Developers can be more productive if they focus on creating Java classes that implement business logic, and use native Java classes to represent data from the data sources. Mapping between the Java classes and the data source, if necessary, can be done by an EIS domain expert.

JDO defines interfaces and classes to be used by application programmers when using classes whose instances are to be stored in persistent storage (persistence-capable classes), and specifies the contracts between suppliers of persistence-capable classes and the runtime environment (which is part of the JDO Implementation).

The supplier of the JDO Implementation is hereinafter called the JDO vendor.

1.1 Overview

There are two major objectives of the JDO architecture: first, to provide application programmers a transparent Java-centric view of persistent information, including enterprise data and locally stored data; and second, to enable pluggable implementations of data-stores into application servers.

The Java Data Objects architecture defines a standard API to data contained in local storage systems and heterogeneous enterprise information systems, such as ERP, mainframe transaction processing and database systems. The architecture also refers to the Connector architecture[see Appendix A reference 4] which defines a set of portable, scalable, secure, and transactional mechanisms for the integration of EIS with an application server.

This architecture enables a local storage expert, an enterprise information system (EIS) vendor, or an EIS domain expert to provide a standard data view (JDO Implementation) for the local data or EIS.

1.2 Scope

The JDO architecture defines a standard set of contracts between an application programmer and an JDO vendor. These contracts focus on the view of the Java instances of persistence-capable classes.

JDO uses the Connector Architecture [see Appendix A reference 4] to specify the contract between the JDO vendor and an application server. These contracts focus on the important aspects of integration with heterogeneous enterprise information systems: instance management, connection management, and transaction management.

To provide transparent storage of local data, the JDO architecture does not require the Connector Architecture in non-managed (non-application server) environments.

1.3 Target Audience

The target audience for this specification includes:

- application developers
- JDO vendors
- enterprise information system (EIS) vendors and EIS Connector providers
- container providers
- enterprise system integrators
- enterprise tool vendors

*JDO defines two types of interfaces: the **JDO API**, of primary interest to application developers (the JDO instance life cycle) and the **JDO SPI**, of primary interest to container providers and JDO vendors. An italicized notice may appear at the end of a section, directing readers interested only in the API side to skip to the next API-side section.*

1.4 Organization

This document describes the rationale and goals for a standard architecture for specifying the interface between an application developer and a local file system or EIS datastore. It then elaborates the JDO architecture and its relationship to the Connector architecture.

The document next describes two typical JDO scenarios, one managed (application server) and the other non-managed (local file storage). This chapter explains key roles and responsibilities involved in the development and deployment of portable Java applications that require persistent storage.

The document then details the prescriptive aspects of the architecture. It starts with the JDO instance, which is the application programmer-visible part of the system. It then details the JDO `PersistenceManager`, which is the primary interface between a persistence-aware application, focusing on the contracts between the application developer and JDO implementation provider. Finally, the contracts for connection and transaction management between the JDO vendor and application server vendor are defined.

1.5 Document Convention

A Palatino font is used for describing the JDO architecture.

A courier font is used for code fragments.

1.6 Terminology Convention

“Must” is used where the specified component is required to implement some interface or action to be compliant with the specification.

“Might” is used where there is an implementation choice whether or how to implement a method or function.

“Should” is used to describe objectives of the specification and recommended application programming usage. If the recommended usage is not followed by applications, behavior is non-portable, unexpected, or unspecified.

“Should” is also used where there is a recommended choice for possibly different implementation actions. If the recommended usage is not followed by implementations, inefficiencies might result.

2 Overview

This chapter introduces key concepts that are required for an understanding of the JDO architecture. It lays down a reference framework to facilitate a formal specification of the JDO architecture in the subsequent chapters of this document.

2.1 Definitions

2.1.1 JDO common interfaces

JDO Instance

A JDO instance is a Java programming language instance of a Java class that implements the application functions, and represents data in a local file system or enterprise datastore. Without limitation, the data might come from a single datastore entity, or from a collection of entities. For example, an entity might be a single object from an object database, a single row of a relational database, the result of a relational database query consisting of several rows, a merging of data from several tables in a relational database, or the result of executing a data retrieval API from an ERP system.

JDO instances implement the `PersistenceCapable` interface, either explicitly by the class writer, or implicitly by the results of the enhancer. The objective of JDO is that most user-written classes, including both entity-type classes and utility-type classes, might be persistence capable. The limitations are that the persistent state of the class must be represented entirely by the state of its Java fields, and that the class be enhanced (or otherwise be written to implement the `PersistenceCapable` interface) prior to being loaded into the execution environment of the Java Virtual Machine. Thus, system-type classes such as `System`, `Thread`, `Socket`, `File`, and the like cannot be JDO persistence-capable, but common user-defined classes can be.

JDO Implementation

A JDO implementation is a collection of classes that implement the JDO contracts. The JDO implementation might be provided by an EIS vendor or by a third party vendor, collectively known as JDO vendor. The third party might provide an implementation that is optimized for a particular application domain, or might be a general purpose tool (such as a relational mapping tool, embedded object database, or enterprise object database).

The primary interface to the application is `PersistenceManager`, with interfaces `Query` and `Transaction` playing supporting roles for application control of the execution environment.

JDO Enhancer

A JDO enhancer, or byte code enhancer, is a program that modifies the byte codes of application-component Java class files to enable transparent loading and storing of the fields of their persistent instances. The JDO reference implementation (reference enhancement) contains an approach for the enhancement of Java class files to allow for enhanced class files to be shared among several coresident JDO implementations.

Alternative approaches to byte code enhancement are preprocessing or code generation. If one of the alternatives is used instead of byte code enhancement, the `PersistenceCapable` contract must be implemented.

A JDO implementation is free to extend the Reference Enhancement contract with implementation-specific methods and fields that might be used by its runtime environment.

Binary Compatibility Requirement: classes enhanced by the reference enhancer must be usable by any JDO compliant implementation; classes enhanced by a JDO compliant implementation must be usable by the reference implementation; and classes enhanced by a JDO compliant implementation must be usable by any other JDO compliant implementation.

The following table determines which interface is used by a JDO implementation based on

Table 1: Which Enhancement Interface is Used

	Reference Runtime	Vendor A Runtime	Vendor B Runtime
Reference Enhancer	Reference Enhancement	Reference Enhancement	Reference Enhancement
Vendor A Enhancer	Reference Enhancement	Vendor A Enhancement	Reference Enhancement
Vendor B Enhancer	Reference Enhancement	Reference Enhancement	Vendor B Enhancement

the enhancement of the persistence-capable class. For example, if Vendor A runtime detects that the class was enhanced by its own enhancement, then the runtime will use its enhancement contract. Otherwise, it will use the Reference Enhancement contract.

Readers primarily interested in JDO as a local persistence mechanism can ignore the following section, as it details architectural features not relevant to local environments. Skip to 2.2 – Rationale.

2.1.2 JDO in a managed environment

This discussion provides a bridge to the Connector architecture, which JDO uses for transaction and connection management in application server environments.

Enterprise Information System (EIS)

An EIS provides the information infrastructure for an enterprise. An EIS offers a set of services to its clients. These services are exposed to clients as local and/or remote interfaces. Examples of EIS include:

- relational database system;
- object database system;
- ERP system; and
- mainframe transaction processing system.

EIS Resource

An EIS resource provides EIS-specific functionality to its clients. Examples are:

- a record or set of records in a database system;
- a business object in an ERP system; and
- a transaction program in a transaction processing system

Resource Manager (RM)

A resource manager manages a set of shared resources. A client requests access to a resource manager to use its managed resources. A transactional resource manager can par-

ticipate in transactions that are externally controlled and coordinated by a transaction manager.

Connection

A connection provides connectivity to a resource manager. It enables an application client to connect to a resource manager, perform transactions, and access services provided by that resource manager. A connection can be either transactional or non-transactional. Examples include a database connection and a SAP R/3 connection.

Application Component

An application component can be a server-side component, such as an EJB, JSP, or servlet, that is deployed, managed and executed on an application server. It can be a component executed on the web-client tier but made available to the web-client by an application server, such as a Java applet, or DHTML page. It might also be an embedded component executed in a small footprint device using flash memory for persistent storage.

Session Beans

Session objects are EJB application components that execute on behalf of a single client, might be transaction aware, update data in an underlying datastore, and do not directly represent data in the datastore.

Entity Beans

Entity objects are EJB application components that provide an object view of transactional data in an underlying datastore, allow shared access from multiple users, including session objects and remote clients, and directly represent data in the datastore.

Helper objects

Helper objects are application components that provide an object view of data in an underlying datastore, allow transactionally consistent view of data in multiple transactions, are usable by local session and entity beans, but do not have a remote interface.

Container

A container is a part of an application server that provides deployment and runtime support for application components. It provides a federated view of the underlying application server services for the application components. For more details on different types of standard containers, refer to Enterprise JavaBeans (EJB) [see Appendix A reference 1], Java Server Pages (JSP), and Servlets specifications.

2.2 Rationale

There is no existing Java platform specification that proposes a standard architecture for storing the state of Java objects persistently in transactional datastores.

The JDO architecture offers a Java solution to the problem of presenting a consistent view of data from the large number of application programs and enterprise information systems already in existence. By using the JDO architecture, it is not necessary for application component vendors to customize their products for each type of datastore.

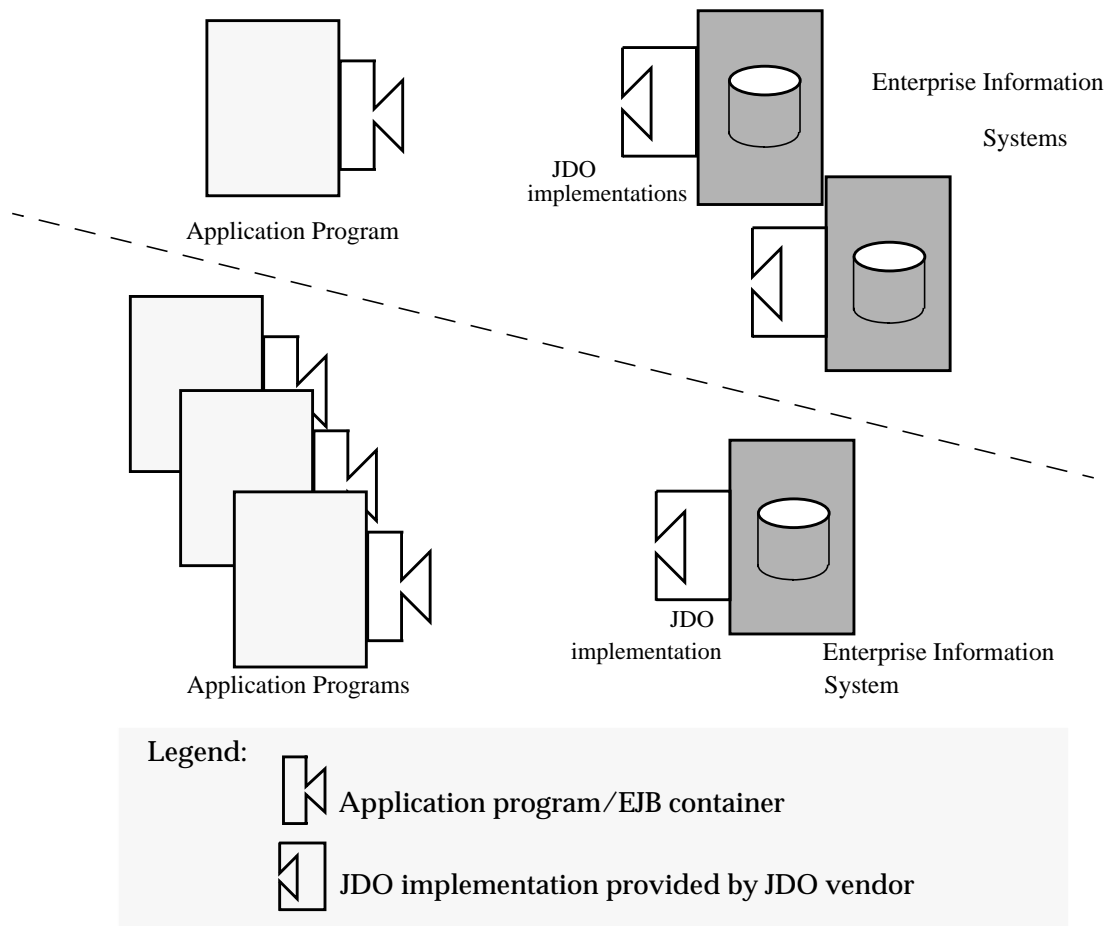
This architecture enables an EIS vendor to provide a standard data access interface for its EIS. The JDO implementation is plugged into an application server and provides underlying infrastructure for integration between the EIS and application components.

Similarly, a third party vendor can provide a standard data access interface for locally managed data such as would be found in an embedded device.

An application component vendor extends its system only once to support the JDO architecture and then exploits multiple data sources. Likewise, an EIS vendor provides one standard JDO implementation and it has the capability to work with any application component that uses the JDO architecture.

The Figure 1.0 on page 21 shows that an application component can plug into multiple JDO implementations. Similarly, multiple JDO implementations for different EISes can plug into an application component. This standard plug-and-play is made possible through the JDO architecture.

Figure 1.0 Standard plug-and-play between application programs and EISes using JDO



2.3 Goals

The JDO architecture has been designed with the following goals:

- The JDO architecture provides a transparent interface for application component and helper class developers to store data without learning a new data access language for each type of persistent data storage.

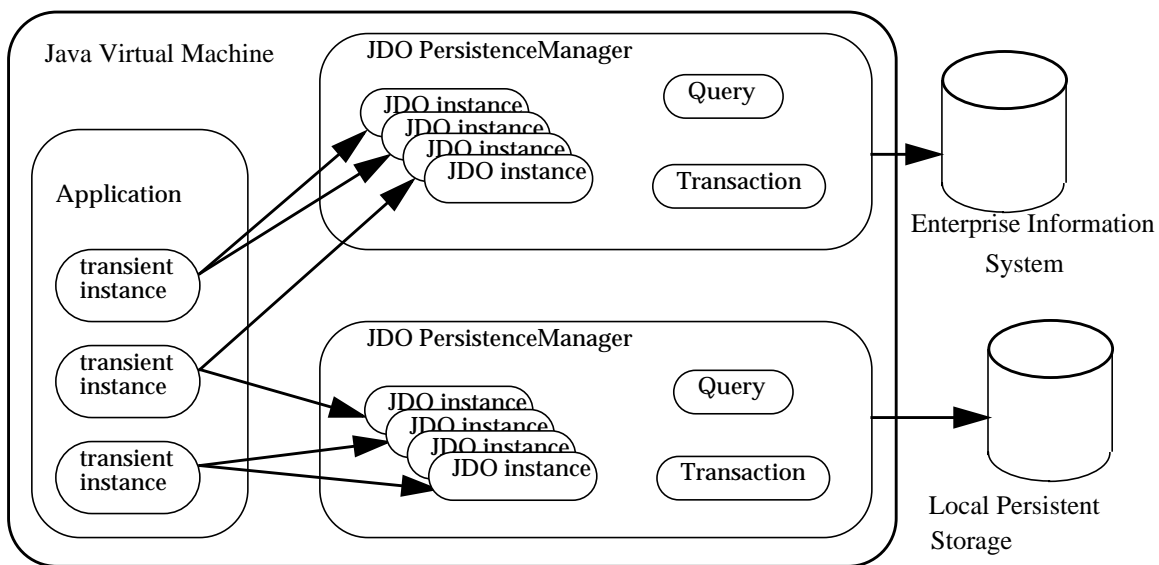
- The JDO architecture simplifies the development of scalable, secure and transactional JDO implementations for a wide range of EISes — ERP systems, database systems, mainframe-based transaction processing systems.
- The JDO architecture is implementable for a wide range of heterogeneous local file systems and EISes. The intent is that there will be various implementation choices for different EIS—each choice based on possibly application-specific characteristics and mechanisms of a mapping to an underlying EIS.
- The JDO architecture is suitable for a wide range of uses from embedded small footprint systems to large scale enterprise application servers. This architecture provides for exploitation of critical performance features from the underlying EIS, such as query evaluation and relationship management.
- The JDO architecture uses the J2EE Connector Architecture to make it applicable to all J2EE platform compliant application servers from multiple vendors.
- The JDO architecture makes it easy for application component developers to use the Java programming model to model the application domain and transparently retrieve and store data from various EIS systems.
- The JDO architecture defines contracts and responsibilities for various roles that provide pieces for standard connectivity to an EIS. This enables a standard JDO implementation from a EIS or third party vendor to be pluggable across multiple application servers.
- The connector architecture also enables an application programmer in a non-managed application environment to directly use the JDO implementation to access the underlying file system or EIS. This is in addition to a managed access to an EIS with the JDO implementation deployed in the middle-tier application server. In the former case, application programmers will not rely on the services offered by a middle-tier application server for security, transaction, and connection management, but will be responsible for managing these system-level aspects by using the EIS connector.

3 JDO Architecture

3.1 Overview

Multiple JDO implementations - possibly multiple implementations per type of EIS or local storage - are pluggable into an application server or usable directly in a two tier or embedded architecture. This enables application components, deployed either on a middle-tier application server or on a client-tier, to access the underlying datastores using a consistent Java-centric view of data. The JDO implementation provides the necessary mapping from Java objects into the special data types and relationships of the underlying datastore.

Figure 2.0 Overview of non-managed JDO architecture



In a non-managed environment, the JDO implementation hides the EIS specific issues such as data type mapping, relationship mapping, and data retrieval and storage. The application component sees only the Java view of the data organized into classes with relationships and collections presented as native Java constructs.

Managed environments additionally provide transparency for the application components' use of system-level mechanisms - distributed transactions, security, and connection management, by hiding the contracts between the application server and JDO implementations.

With both managed and non-managed environments, an application component developer focuses on the development of business and presentation logic for the application components without getting involved in the issues related to connectivity with a specific EIS.

3.2 JDO Architecture

3.2.1 Two tier usage

For simple two tier usage, JDO exposes to the application component two primary interfaces: `javax.jdo.PersistenceManager`, from which services are requested; and `javax.jdo.spi.PersistenceCapable`, which provides the management view of user-defined persistence-capable classes.

The `PersistenceManager` interface provides services such as query management, transaction management, and life cycle management for instances of persistence-capable classes.

The `PersistenceCapable` interface provides services such as life cycle state management for instances of persistence capable classes.

Readers primarily interested in JDO as a local persistence mechanism can ignore the following sections. Skip to 4 – Roles and Scenarios.

3.2.2 Application server usage

For application server usage, the JDO architecture uses the J2EE Connector architecture, which defines a standard set of system-level contracts between the application server and EIS connectors. These system-level contracts are implemented in a resource adapter from the EIS side.

The JDO persistence manager is a caching manager as defined by the J2EE Connector architecture, that might use either its own (native) resource adapter or a third party resource adapter. If the JDO `PersistenceManager` has its own resource adapter, then implementations of the system-level contracts specified in the J2EE Connector architecture must be provided by the JDO vendor. These contracts include `ManagedConnectionFactory`, `XAResource`, and `LocalTransaction` interfaces.

The JDO `Transaction` must implement the `Synchronization` interface so that transaction completion events can cause flushing of state through the underlying connector to the EIS.

The application components are unable to distinguish between JDO implementations that use native resource adapters and JDO implementations that use third party resource adapters. However, the deployer will need to understand that there are two configurable components: the JDO `PersistenceManager` and its underlying resource adapter.

For convenience, the `PersistenceManagerFactory` provides the interface necessary to configure the underlying resource adapter.

Resource Adapter

A resource adapter provided by the JDO vendor is called a native resource adapter, and the interface is specific to the JDO vendor. It is a system-level software driver that is used by an application server or an application client to connect to a resource manager.

The resource adapter plugs into a container (provided by the application server). The application components deployed on the container then use the client API exposed by `javax.jdo.PersistenceManager` to access the JDO `PersistenceManager`. The JDO implementation in turn uses the underlying resource adapter interface specific to the data-

store. The resource adapter and application server collaborate to provide the underlying mechanisms - transactions, security and connection pooling - for connectivity to the EIS.

The resource adapter is located within the same VM as the JDO implementation using it. Examples of JDO native resource adapters are:

- Object/Relational (O/R) products that use their own native drivers to connect to object relational databases
- Object Database (OODBMS) products that store Java objects directly in object databases

Examples of non-native resource adapter implementations are:

- O/R mapping products that use JDBC drivers to connect to relational databases
- Hierarchical mapping products that use mainframe connectivity tools to connect to hierarchical transactional systems

Pooling

There are two levels of pooling in the JDO architecture. JDO `PersistenceManagers` might be pooled, and the underlying connections to the datastores might be independently pooled.

Pooling of the connections is governed by the Connector Architecture contracts. Pooling of `PersistenceManagers` is an optional feature of the JDO Implementation, and is not standardized for two-tier applications. For managed environments, `PersistenceManager` pooling is required to maintain correct transaction associations with `PersistenceManagers`.

For example, a JDO `PersistenceManager` instance might be bound to a session running a long duration optimistic transaction. This instance cannot be used by any other user for the duration of the optimistic transaction.

During the execution of a business method associated with the session, a connection might be required to fetch data from the datastore. The `PersistenceManager` will request a connection from the connection pool to satisfy the request. Upon termination of the business method, the connection is returned to the pool but the `PersistenceManager` remains bound to the session.

After completion of the optimistic transaction, the `PersistenceManager` instance might be returned to the pool and reused for a subsequent transaction.

Contracts

JDO specifies the application level contract between the application components and the JDO `PersistenceManager`.

The J2EE Connector architecture specifies the standard contracts between application servers and an EIS connector used by a JDO implementation. These contracts are required for a JDO implementation to be used in an application server environment. The Connector architecture defines important aspects of integration: connection management, transaction management, and security.

The connection management contracts are implemented by the EIS resource adapter (which might include a JDO native resource adapter).

The transaction management contract is between the transaction manager (logically distinct from the application server) and the connection manager. It supports distributed transactions across multiple application servers and heterogeneous data management programs.

The security contract is required for secure access by the JDO connection to the underlying datastore.

Figure 3.0 Contracts between application server and native JDO resource adapter

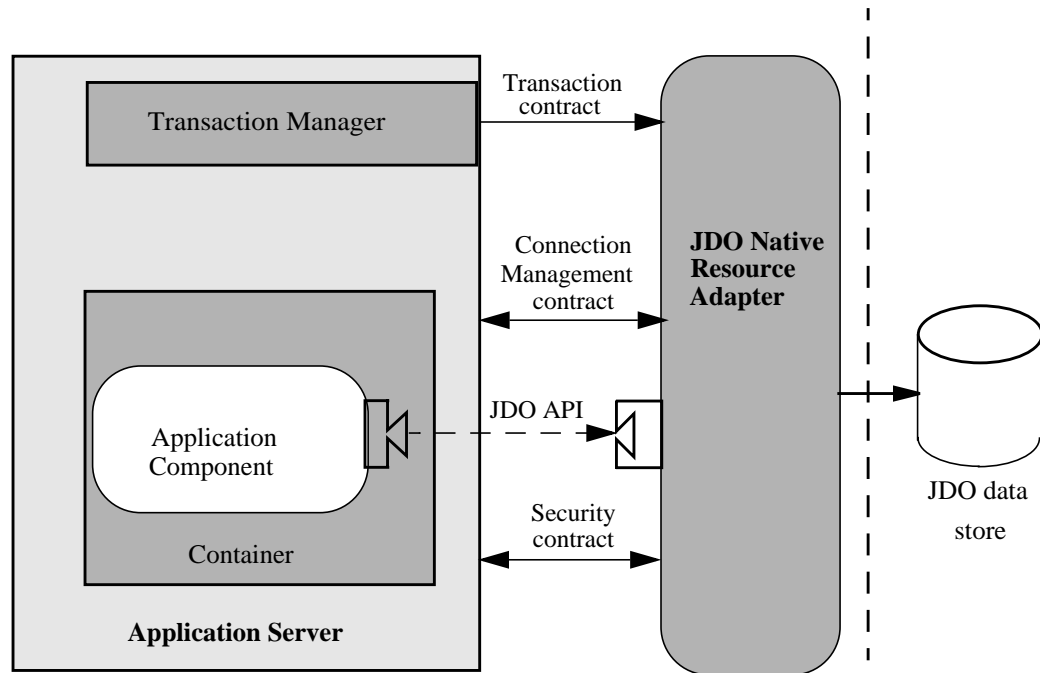
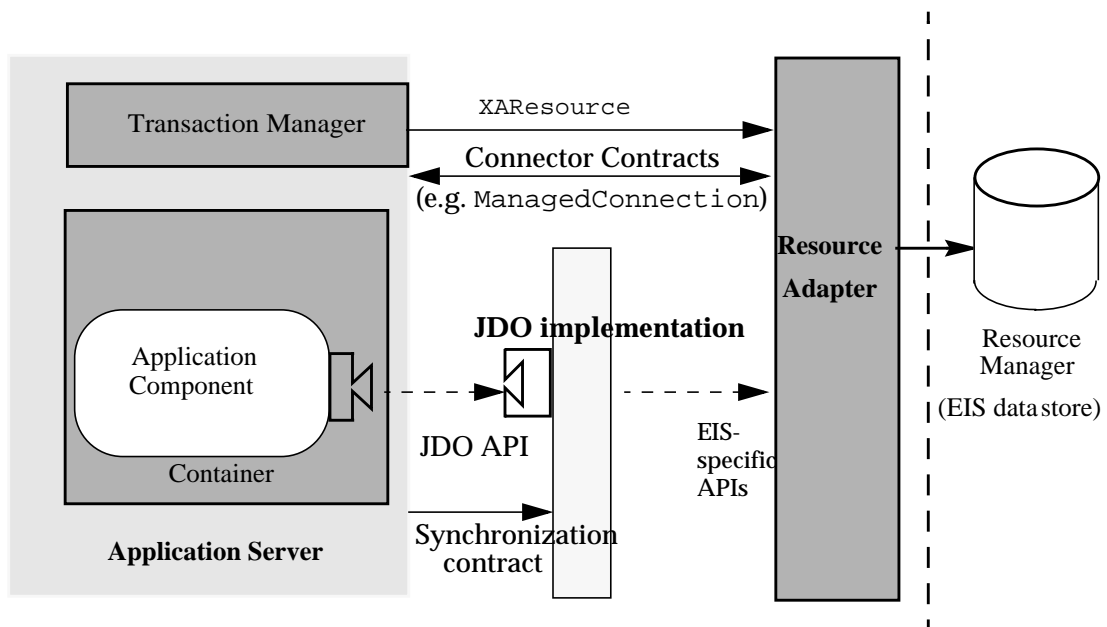


Figure 4.0 Contracts between application server and layered JDO implementation



The above diagram illustrates the relationship between a JDO implementation provided by a third party vendor and an EIS-provided resource adapter.

4 Roles and Scenarios

4.1 Roles

This chapter describes roles required for the development and deployment of applications built using the JDO architecture. The goal is to identify the nature of the work specific to each role so that the contracts specific to each role can be implemented on each side of the contracts.

The detailed contracts are specified in other chapters of this specification. The specific intent here is to identify the primary users and implementors of these contracts.

4.1.1 Application Developer

The application developer writes software to the **JDO API**. The JDO application developer does not have to be an expert in the technology related to a specific datastore.

4.1.2 Application Component Provider

The application component provider produces an application library that implements application functionality through Java classes with business methods that store data persistently in one or more EISes through the JDO API.

There are two types of application components that interact with JDO. JDO-transparent application components, typically helper classes, are those that use JDO to have their state stored in a transactional datastore, and directly access other components by references of their fields. Thus, they do not need to use JDO APIs directly.

JDO-aware application components (bean-managed persistent entity beans and session beans) use services of JDO by directly accessing its API. These components use JDO query facilities to retrieve collections of JDO instances from the datastore, make specific instances persistent in a particular datastore, delete specific persistent instances from the datastore, interrogate the cached state of JDO instances, or explicitly manage the cache of the JDO `PersistenceManager`. These application components are non-transparent users of JDO.

Session beans that use helper JDO classes interact directly with `PersistenceManager` and `PersistenceCapable`. Entity beans might be generated by tools that analyze `PersistenceCapable` classes and produce bean-managed persistence beans corresponding to some of them. These generated entity beans use the standard JDO `PersistenceManager` contracts to request services.

The output of the application component provider is a set of jar files containing application components.

4.1.3 Application Assembler

The application assembler is a domain expert who assembles application components from multiple sources including in-house developers and application library vendors. The application assembler can combine different types of application components, for example EJBs, servlets, or JSPs, into a single end-user-visible application.

The input of the application assembler is one or more jar files, produced by application component providers. The output is one or more jar files with deployment specific descriptions.

4.1.4 Deployer

The deployer is responsible for configuring assembled components into specific operational environments. The deployer resolves all external references from components to other components or to the operational system.

For example, the deployer will bind application components in specific operating environments to datastores in those environments, and will resolve references from one application component to another. This typically involves using container-provided tools.

The deployer must understand, and be able to define, security roles, transactions, and connection pooling protocols for multiple datastores, application components, and containers.

4.1.5 System Administrator

The system administrator manages the configuration and administration of multiple containers, resource adapters and EISs that combine into an operational system.

Readers primarily interested in developing applications with the JDO API can ignore the following sections. Skip to 4.2 – Scenario: Embedded calendar management system.

4.1.6 JDO Vendor

The JDO vendor is an expert in the technology related to a specific datastore and is responsible for providing a **JDO SPI** implementation for that specific datastore. Since this role is highly datastore specific, a datastore vendor will often provide the standard JDO implementation.

A vendor can also provide a JDO implementation and associated set of application development tools through a loose coupling with a specific third party datastore. Such providers specialize in writing connectors and related tools for a specific EIS or might provide a more general tool for a large number of datastores.

The JDO vendor requires that the EIS vendor has implemented the J2EE Connector architecture and the role of the JDO implementation is that of a synchronization adapter to the connector architecture.

Readers primarily interested in JDO as a local persistence mechanism can ignore the following section. Skip to 4.2 – Scenario: Embedded calendar management system.

4.1.7 Connector Provider

The connector provider is typically the vendor of the EIS or datastore, and is responsible for supplying a library of interface implementations that satisfy the resource adapter interface.

In the JDO architecture, the Connector is a separate component, supplied by either the JDO vendor or by an EIS vendor or third party.

4.1.8 Application Server Vendor

An application server vendor [see Appendix A reference 1], [see Appendix A reference 3] provides an implementation of a J2EE compliant application server that provides support for component-based enterprise applications. A typical application server vendor is an OS vendor, middleware vendor, or database vendor.

The role of application server vendor will typically be the same as that of the container provider.

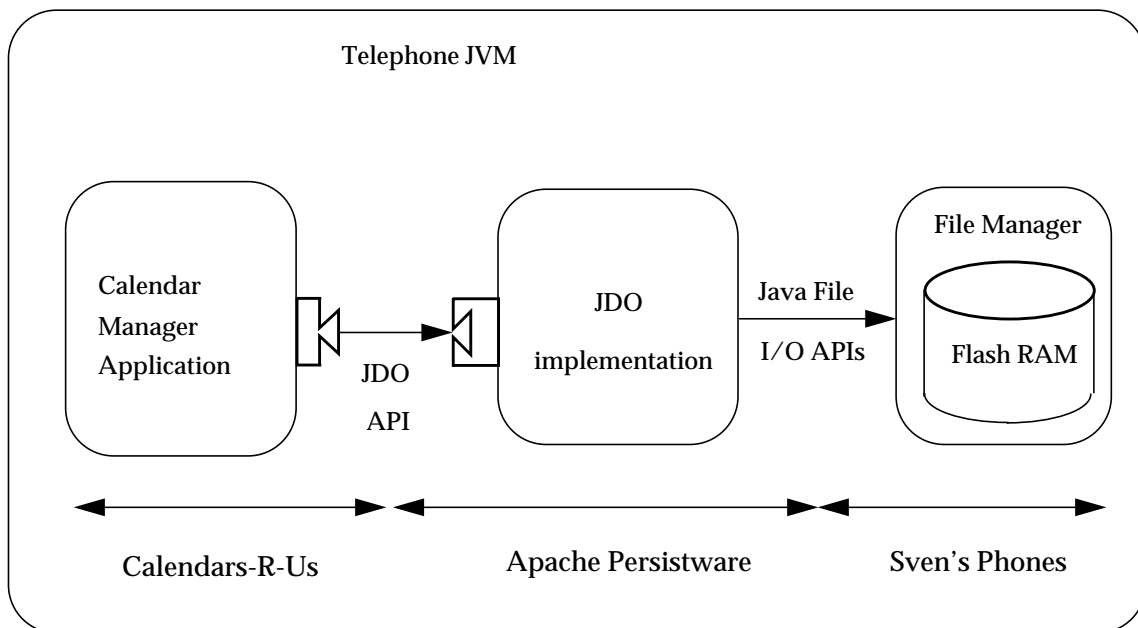
4.1.9 Container Provider

For bean-managed persistence, the container provides deployed application components with transaction and security management, distribution of clients, scalable management of resources and other services that are generally required as part of a managed server platform.

4.2 Scenario: Embedded calendar management system

This section describes a scenario to illustrate the use of JDO architecture in an embedded mobile device such as a personal information manager (PIM) or telephone.

Figure 5.0 Scenario: Embedded calendar manager



Sven's Phones is a manufacturer of high function telephones for the traveling businessperson. They have implemented a Java operating environment that provides persistence via a Java file I/O subsystem that writes to flash RAM.

Apache Persistware is a supplier of JDO software that has a small footprint and as such, is especially suited for embedded devices such as personal digital assistants and telephones. They use Java file I/O to store JDO instances persistently.

Calendars-R-Us is a supplier of appointment and calendar software that is written for several operating environments, from high function telephones to desktop workstations and enterprise application servers.

Calendars-R-Us uses the JDO API directly to manage calendar appointments on behalf of the user. The calendar application needs to insert, delete, and change calendar appointments based on the user's keypad input. It uses Java application domain classes: Ap-

pointment, Contact, Note, Reminder, Location, and TelephoneNumber. It employs JDK library classes: Time, Date, ArrayList, and Calendar.

Calendars-R-Us previously used Java file I/O APIs directly, but ran into several difficulties. The most efficient storage for some environments was an indexed file system, which was required only for management of thousands of entries. However, when they ported the application to the telephone, the indexed file system was too resource-intensive, and had to be abandoned.

They then wrote a data access manager for sequential files, but found that it burned out the flash RAM due to too much rewriting of data. They concluded that they needed to use the services of another software provider who specialized in persistence for flash RAM in embedded devices.

Apache Persistware developed a file access manager based on the Berkeley File System and successfully sold it to a range of Java customers from embedded devices to workstations. The interface was proprietary, which meant that every new sale was a challenge, because customers were loath to invest resources in learning a different interface for each environment they wanted to support. After all, Java was portable. Why wasn't file access?

Sven's Phones was a successful supplier of telephones to the mobile professional, but found themselves constrained by a lack of software developers. They wanted to offer a platform on which specially tailored software from multiple vendors could operate, and take advantage of external developers to write software for their telephones.

The solution to all of these issues was to separate the software into components that could be tailored by the domain expert for each component.

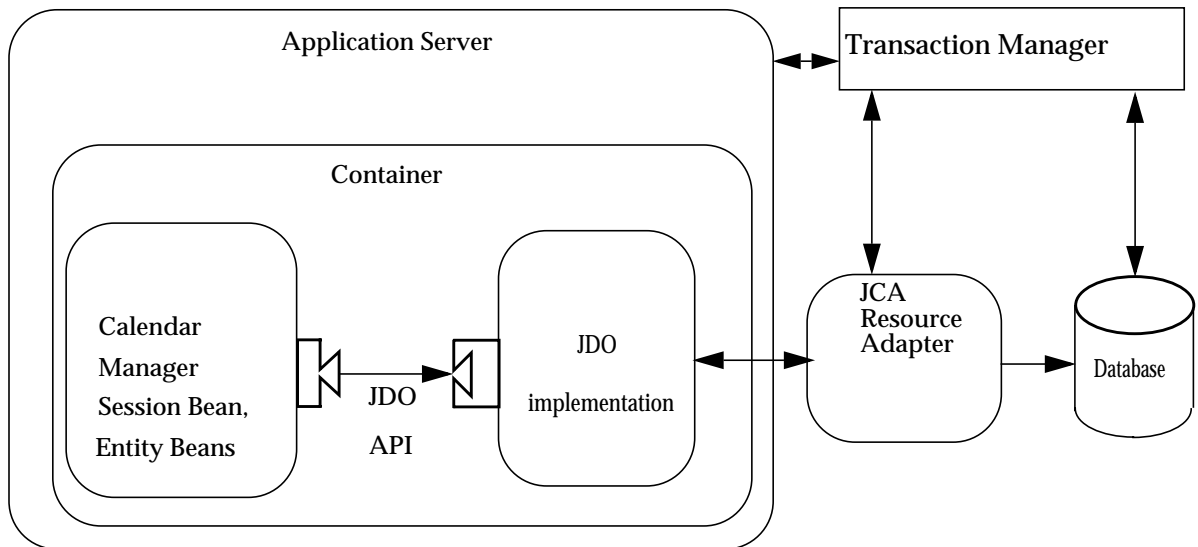
Sven's phones implemented the Java runtime environment for their phones, and wrote an efficient sequential file I/O manager that implemented the Java file I/O interface. This interface was used by Apache Persistware to build a JDO implementation, including a JDO instance handler and a JDO query manager.

Using the JDO interface, Calendars-R-Us rewrote just the query part of their software. The application classes did not have to be changed. Only the persistence interface that queried for specific instances needed to be modified.

Readers primarily interested in JDO as a local persistence mechanism can ignore the following section. Skip to 5 – Life Cycle of JDO Instances.

4.3 Scenario: Enterprise Calendar Manager

Calendars-R-Us also supports workstations and enterprise mainframes with their calendar software, and they use the same interface for persistence in all environments. For enterprise environments, they simply need to use a different JDO implementation supplied by a different vendor to achieve persistence for their calendar objects.

Figure 6.0 Scenario: Enterprise Calendar Manager

In this scenario, the JDO implementation is provided by a vendor that maps Java objects to relational databases. The implementation uses a JCA Resource Adapter to connect to the datastore.

The JDO `PersistenceManager` is a caching manager, as defined by the Connector architecture, and it is configured to use a JCA Resource Adapter. The `PersistenceManager` instance might be cached when used with a Session Bean, and might be serially reused for multiple session beans.

Multiple JDO `PersistenceManager` instances might serially reuse connections from the same pool of JDBC drivers. Therefore, resource sharing is accomplished while maintaining state for each session.

5 Life Cycle of JDO Instances

This chapter specifies the life cycle for persistence capable class instances, hereinafter “JDO instances”. The classes include behavior as specified by the class (bean) developer, and additional behavior as provided by the reference enhancer or JDO vendor’s deployment tool. The enhancement of the classes allows application developers to treat JDO instances as if they were normal instances, with automatic fetching of persistent state from the JDO implementation.

5.1 Overview

JDO instances might be either transient or persistent. That is, they might represent the persistent state of data contained in a transactional datastore. If a JDO instance is transient (and not transactional), then there is no difference between its behavior and the behavior of an instance of the unmodified (unenhanced) persistence capable class.

If a JDO instance is persistent, its behavior is linked to the transactional datastore with which it is associated. The JDO implementation automatically tracks changes made to the values in the instance, and automatically refreshes values from the datastore and saves values into the datastore as required to preserve transactional integrity of the data. Persistent instances stored in the datastore retain their class and the state of their persistent fields. Changing the class of a persistent instance is not supported explicitly by the JDO API. However, it might be possible for an instance to change class based on external modifications to the datastore.

During the life of a JDO instance, it transitions among various states until it is finally garbage collected by the JVM. During its life, the state transitions are governed by the behaviors executed on it directly as well as behaviors executed on the JDO `PersistenceManager` by both the application and by the execution environment (including the `TransactionManager`).

During the life cycle, instances at times might be inconsistent with the datastore as of the beginning of the transaction. If instances are inconsistent, the notation for that instance in JDO is “dirty”. Instances made newly persistent, deleted, or modified in the transaction are dirty.

At times, the JDO implementation might store the state of persistent instances in the datastore. This process is called “flushing”, and it does not affect the “dirty” state of the instances.

Under application control, transient JDO instances might observe transaction boundaries, in which the state of the instances is either preserved (on commit) or restored (on rollback). Transient instances that observe transaction boundaries are called transient transactional instances. Support for transient transactional instances is a JDO option; that is, a JDO compliant implementation is not required to implement the APIs that cause the state transitions associated with transient transactional instances.

Under application control, persistent JDO instances might not observe transaction boundaries. These instances are called persistent-nontransactional instances, and the life cycle of

these instances is not affected by transaction boundaries. Support for nontransactional instances is a JDO option.

If a JDO instance is persistent or transactional, it contains a non-null reference to a JDO `StateManager` instance which is responsible for managing the JDO instance state changes and for interfacing with the JDO `PersistenceManager`.

5.2 Goals

The JDO instance life cycle has the following goals:

- The fact of persistence should be transparent to both JDO instance developer and application component developer
- JDO instances should be able to be used efficiently in a variety of environments, including managed (application server) and non-managed (two-tier) cases
- Several JDO `PersistenceManagers` might be coresident and might share the same persistence capable classes (although a JDO instance can be associated with only one `PersistenceManager` at a time)

5.3 Architecture:

JDO Instances

For transient JDO instances, there is no supporting infrastructure required. That is, transient instances will never make calls to methods to the persistence infrastructure. There is no requirement to instantiate objects outside the application domain. There is no difference in behavior between transient instances of enhanced classes and transient instances of the same non-enhanced classes, with some exceptions:

- additional methods and fields added by the enhancement process are visible to Java core reflection,
- timing of method execution is different because of added byte codes,
- extra methods for registration of metadata are executed at class load time.

Persistent JDO instances execute in an environment that contains an instance of the JDO `PersistenceManager` responsible for its persistent behavior. The JDO instance contains a reference to an instance of the JDO `StateManager` responsible for the state transitions of the instance as well as for managing the contents of the fields of the instance. The `PersistenceManager` and the `StateManager` might be implemented by the same instance, but their interfaces are distinct.

The contract between the persistence capable class and other application components extends the contract between the associated non-persistence capable class and application components. These contract extensions support interrogation of the life cycle state of the instances and are intended for use by management parts of the system.

JDO State Manager

Persistent and transactional JDO instances contain a reference to a JDO `StateManager` instance to which all of the JDO interrogatives are delegated. The associated JDO `StateManager` instance maintains the state changes of the JDO instance and interfaces with the JDO `PersistenceManager` to manage the values of the datastore.

JDO Managed Fields

Only some fields are of interest to the persistence infrastructure: fields whose values are stored in the datastore are called persistent; fields that participate in transactions (their values may be restored during rollback) are called transactional; fields of either type are called managed.

5.4 JDO Identity

Java defines two concepts for determining if two instances are the same instance (identity), or represent the same data (equality). JDO extends these concepts to determine if two in-memory instances represent the same stored object.

Java object identity is entirely managed by the Java Virtual Machine. Instances are identical if and only if they occupy the same storage location within the JVM.

Java object equality is determined by the class. Distinct instances are equal if they represent the same data, such as the same value for an `Integer`, or same set of bits for a `BitSet`.

The interaction between Java object identity and equality is an important one for JDO developers. Java object equality is an application specific concept, and JDO implementations must not change the application's semantic of equality. Still, JDO implementations must manage the cache of JDO instances such that there is only one JDO instance associated with each `JDO PersistenceManager` representing the persistent state of each corresponding datastore object. Therefore, JDO defines object identity differently from both the Java VM object identity and from the application equality.

Applications should implement `equals` for persistence-capable classes differently from `Object`'s default `equals` implementation, which simply uses the Java VM object identity. This is because the JVM object identity of a persistent instance cannot be guaranteed between `PersistenceManagers` and across space and time, except in very specific cases noted below.

Additionally, if persistence instances are stored in the datastore and are queried using the `==` query operator, or are referred to by a persistent collection that enforces equality (`Set`, `Map`) then the implementation of `equals` should exactly match the JDO implementation of equality, using the primary key or `ObjectId` as the key. This policy is not enforced, but if it is not correctly implemented, semantics of standard collections and JDO collections may differ.

To avoid confusion with Java object identity, this document refers to the JDO concept as JDO identity.

Three Types of JDO identity

JDO defines three types of JDO identity:

- Application identity - JDO identity managed by the application and enforced by the datastore; JDO identity is often called the primary key
- Datastore identity - JDO identity managed by the datastore without being tied to any field values of a JDO instance
- Nondurable identity - JDO identity managed by the implementation to guarantee uniqueness in the JVM but not in the datastore

The type of JDO identity used is a property of a `JDO PersistenceCapable` class and is fixed at enhancement time.

The representation of JDO identity in the JVM is via a JDO object id. Every persistent instance (Java instance representing a persistent object) has a corresponding object id. There might be an instance in the JVM representing the object id, or not. The object id JVM instance corresponding to a persistent instance might be acquired by the application at run time and used later to obtain a reference to the same datastore object, and it might be saved to and retrieved from durable storage (by serialization or other technique).

The class representing the object id for datastore and nondurable identity classes is defined by the JDO implementation. The implementation might choose to use any class that satisfies the requirements for the specific type of JDO identity for a class. It might choose the same class for several different JDO classes, or might use a different class for each JDO class.

The class representing the object id for application identity classes is defined by the application in the metadata, and might be provided by the application or by a JDO vendor tool.

The application-visible representation of the JDO identity is an instance that is completely under the control of the application. The object id instances used as parameters or returned by methods in the JDO interface (`getObjectId`, `getTransactionalObjectId`, and `getObjectById`) will never be saved internally; rather, they are copies of the internal representation or used to find instances of the internal representation.

Therefore, the object returned by any call to `getObjectId` might be modified by the user, but that modification does not affect the identity of the object that was originally referred. That is, the call to `getObjectId` returns only a copy of the object identity used internally by the implementation.

It is a requirement that the instance returned by a call to `getObjectById(Object)` of different `PersistenceManager` instances returned by the same `PersistenceManagerFactory` represent the same persistent object, but with different Java object identity (specifically, all instances returned by `getObjectId` from the instances must return `true` to `equals` comparisons with all others).

Further, any instances returned by any calls to `getObjectById(Object)` with the same object id instance to the same `PersistenceManager` instance must be identical (assuming the instances were not garbage collected between calls).

The JDO identity of transient instances is not defined. Attempts to get the object id for a transient instance will return `null`.

Uniquing

JDO identity of persistent instances is managed by the implementation. For a durable JDO identity (datastore or application), there is only one persistent instance associated with a specific datastore object per `PersistenceManager` instance, regardless of how the persistent instance was put into the cache:

- `PersistenceManager.getObjectById(Object oid, boolean validate);`
- query via a `Query` instance associated with the `PersistenceManager` instance;
- navigation from a persistent instance associated with the `PersistenceManager` instance;
- `PersistenceManager.makePersistent(Object pc);`

Change of identity

Change of identity is supported only for application identity, and is an optional feature of a JDO implementation. An application attempt to change the identity of an instance (by writing a primary key field) where the implementation does not support this optional feature results in `JDOUnsupportedOperationException` being thrown.

NOTE: Application developers should take into account that changing primary key values changes the identity of the instance in the datastore. In production environments where audit trails of changes are kept, change of the identity of datastore instances might cause loss of audit trail integrity, as the historical record of changes does not reflect the current identity in the datastore.

JDO instances using application identity may change their identity during a transaction if the application changes a primary key field. In this case, there is a new JDO Identity associated with the JDO instance immediately upon completion of the statement that changes a primary key field. If a JDO instance is already associated with the new JDO Identity, then a `JDOUserException` is thrown and the statement that attempted to change the primary key field does not complete.

Upon successful commit of the transaction, the existing datastore instance will have been updated with the changed values of the primary key fields.

JDO Identity Support

A JDO implementation is required to support either or both of application (primary key) identity or datastore identity, and may optionally support nondurable identity.

5.4.1 Application (primary key) identity

This is the JDO identity type used for datastores in which the value(s) in the instance determine the identity of the object in the datastore. Thus, JDO identity is managed by the application. The class provided by the application that implements the JDO object id has all of the characteristics of an RMI remote object, making it possible to use the JDO object id class as the EJB primary key class. Specifically:

- the `ObjectId` class must be public;
- the `ObjectId` class must implement `Serializable`;
- the `ObjectId` class must have a public no-arg constructor, which might be the default constructor;
- the field types of all non-static fields in the `ObjectId` class must be serializable, and for portability should be primitive, `String`, `Date`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigDecimal`, or `BigInteger` types; JDO implementations are required to support these types and might support other reference types;
- all serializable non-static fields in the `ObjectId` class must be public;
- the names of the non-static fields in the `ObjectId` class must include the names of the primary key fields in the JDO class, and the types of the corresponding fields must be identical;
- the `equals()` and `hashCode()` methods of the `ObjectId` class must use the value(s) of all the fields corresponding to the primary key fields in the JDO class;
- if the `ObjectId` class is an inner class, it must be static;

- the `ObjectId` class must override the `toString()` method defined in `Object`, and return a `String` that can be used as the parameter of a constructor;
- the `ObjectId` class must provide a `String` constructor that returns an instance that compares equal to an instance that returned that `String` by the `toString()` method.

These restrictions allow the application to construct an instance of the primary key class providing values only for the primary key fields, or alternatively providing only the result of `toString()` from an existing instance. The JDO implementation is permitted to extend the primary key class to use additional fields, not provided by the application, to further identify the instance in the datastore. Thus, the JDO object id instance returned by an implementation might be a subclass of the user-defined primary key class. Any JDO implementation must be able to use the JDO object id instance from any other JDO implementation.

A primary key identity is associated with a specific set of fields. The fields associated with the primary key are a property of the persistence-capable class, and cannot be changed after the class is enhanced for use at runtime. When a transient instance is made persistent, the implementation uses the values of the fields associated with the primary key to construct the JDO identity.

A primary key instance must have none of its primary key fields set to `null` when used to find a persistent instance. The persistence manager will throw `JDOUserException` if the primary key instance contains any `null` values when the key instance is the parameter of `getObjectById`.

Persistence-capable classes that use application identity have special considerations for inheritance. To be portable, the key class must be the same for all classes in the inheritance hierarchy, and key fields must be declared only in the least-derived (topmost) persistence-capable class in the hierarchy.

5.4.2 Datastore identity

This is the JDO identity type used for datastores in which the identity of the data in the datastore does not depend on the values in the instance. The implementation guarantees uniqueness for all instances.

A JDO implementation might choose one of the primitive wrapper classes as the `ObjectId` class (`Short`, `Integer`, `Long`, or `String`), or might choose an implementation-specific class. Implementation-specific classes used as JDO `ObjectId` have the following characteristics:

- the `ObjectId` class must be public;
- the `ObjectId` class must implement `Serializable`;
- the `ObjectId` class must have a public no-arg constructor, which might be the default constructor;
- all serializable fields in the `ObjectId` class must be public;
- the field types of all non-static fields in the `ObjectId` class must be serializable;
- the `ObjectId` class must override the `toString()` method defined in `Object`, and return a `String` that can be used as the parameter of a constructor;
- the `ObjectId` class must provide a `String` constructor that returns an instance that compares equal to an instance that returned that `String` by the `toString()` method.

Note that, unlike primary key identity, datastore identity `ObjectId` classes are **not** required to support equality with `ObjectId` classes from other JDO implementations. Further, the application cannot change the JDO identity of an instance of a class using datastore identity.

5.4.3 Nondurable JDO identity

The primary usage for nondurable JDO identity is for log files, history files, and other similar files, where performance is a primary concern, and there is no need for the overhead associated with managing a durable identity for each datastore instance. Objects are typically inserted into datastores with transactional semantics, but are not accessed by key. They may have references to instances elsewhere in the datastore, but often have no keys or indexes themselves. They might be accessed by other attributes, and might be deleted in bulk.

Multiple objects in the datastore might have exactly the same values, yet an application program might want to treat the objects individually. For example, the application must be able to count the persistent instances to determine the number of datastore objects with the same values. Also, the application might change a single field of an instance with duplicate objects in the datastore, and the expected result in the datastore is that exactly one instance has its field changed. If multiple instances in memory are modified, then instances in the datastore are modified corresponding one-to-one with the modified instances in memory. Similarly, if the application deletes some number of multiple duplicate objects, the same number of the objects in the datastore must be deleted.

As another example, if a datastore instance using nondurable identity is loaded twice into the VM by the same `PersistenceManager`, then two separate instances are instantiated, with two different JDO identities, even though all of the values in the instances are the same. It is permissible to update or delete only one of the instances. At commit time, if only one instance was updated or deleted, then the changes made to that instance are reflected in the datastore by changing the single datastore instance. If both instances were changed, then the transaction will fail at commit, with a `JDOUserException` because the changes must be applied to different datastore instances.

Because the JDO identity is not visible in the datastore, there are special behaviors with regard to nondurable JDO identity:

- the `ObjectId` is not valid after making the associated instance hollow, and attempts to retrieve it will throw a `JDOUserException`;
- the `ObjectId` cannot be used in a different instance of `PersistenceManager` from the one that issued it, and attempts to use it even indirectly (e.g. `getObjectById` with a persistence-capable object as the parameter) will throw a `JDOUserException`;
- the persistent instance might transition to persistent-nontransactional or hollow but cannot transition to any other state afterward;
- attempts to access the instance in the hollow state will throw a `JDOUserException`;
- the results of a query in the datastore will always return instances that are not already in the Java VM, so multiple queries that find the same objects in the datastore will return additional JDO instances with the same values and different JDO identities;
- `makePersistent` will succeed even though another instance already has the same values for all persistent fields.

For JDO identity that is not managed by the datastore, the class that implements `JDO ObjectId` has the following characteristics:

- the `ObjectId` class must be public;
- the `ObjectId` class must have a public constructor, which might be the default constructor or a no-arg constructor;
- all fields in the `ObjectId` class must be public;
- the field types of all fields in the `ObjectId` class must be serializable.

5.5 Life Cycle States

There are ten states defined by this specification. Seven states are required, and three states are optional. If an implementation does not support certain operations, then these three states are not reachable.

Datastore Transactions

The following descriptions apply to datastore transactions with `retainValues=false`. Optimistic transaction and `retainValues=true` state transitions are covered later in this chapter.

5.5.1 Transient (Required)

JDO instances created by using a developer-written constructor that do not involve the persistence environment behave exactly like instances of the unenhanced class.

There is no JDO identity associated with a transient instance.

There is no intermediation to support fetching or storing values for fields. There is no support for demarcation of transaction boundaries. Indeed, there is no transactional behavior of these instances, unless they are referenced by transactional instances at commit time.

When a persistent instance is committed to the datastore, instances referenced by persistent fields of the flushed instance become persistent. This behavior propagates to all instances in the closure of instances through persistent fields. This behavior is called persistence by reachability.

No methods of transient instances throw exceptions except those defined by the class developer.

A transient instance transitions to persistent-new if it is the parameter of `makePersistent`, or if it is referenced by a persistent field of a persistent instance when that instance is committed or made persistent.

5.5.2 Persistent-new (Required)

JDO instances that are newly persistent in the current transaction are persistent-new. This is the state of an instance that has been requested by the application component to become persistent, by using the `PersistenceManager makePersistent` method on the instance.

During the transition from transient to persistent-new

- the associated `PersistenceManager` becomes responsible to implement state interrogation and further state transitions.
- if the transaction flag `restoreValues` is `true`, the values of persistent and transactional non-persistent fields are saved for use during rollback.

- the values of persistent fields of mutable SCO types (e.g. `java.util.Date`, `java.util.HashSet`, etc.) are replaced with JDO implementation-specific copies of the field values that track changes and are owned by the persistent instance.
- a JDO identity is assigned to the instance by the JDO implementation. This identity uniquely identifies the instance inside the `PersistenceManager` and might uniquely identify the instance in the datastore. A copy of the JDO identity will be returned by the `PersistenceManager` method `getObjectId(Object)`.
- instances reachable from this instance by fields of persistence-capable types and collections of persistence-capable types become provisionally persistent and transition from transient to persistent-new. If the instances made provisionally persistent are still reachable at commit time, they become persistent. This effect is recursive, effectively making the transitive closure of transient instances provisionally persistent.

A persistent-new instance transitions to persistent-new-deleted if it is the parameter of `deletePersistent`.

A persistent-new instance transitions to hollow when it is flushed to the datastore during commit when `retainValues` is false. This transition is not visible during `beforeCompletion`, and is visible during `afterCompletion`. During `beforeCompletion`, the user-defined `jdoPreStore` method is called if the class implements `InstanceCallbacks`.

A persistent-new instance transitions to transient at rollback. The instance loses its JDO Identity and its association with the `PersistenceManager`. If `restoreValues` is false, the values of managed fields in the instance are left as they were at the time rollback was called.

5.5.3 Persistent-dirty (Required)

JDO instances that represent persistent data that was changed in the current transaction are persistent-dirty.

A persistent-dirty instance transitions to persistent-deleted if it is the parameter of `deletePersistent`.

Persistent-dirty instances transition to hollow during commit when `retainValues` is false or during rollback when `restoreValues` is false. During `beforeCompletion`, the user-defined `jdoPreStore` method is called if the class implements `InstanceCallbacks`.

If an application modifies a managed field, but the new value is equal to the old value, then it is an implementation choice whether the JDO instance is modified or not. If no modification to any managed field was made by the application, then the implementation must not mark the instance as dirty. If a modification was made to any managed field that changes the value of the field, then the implementation must mark the instance as dirty.

Since changes to array-type fields cannot be tracked by JDO, setting the value of an array-type managed field marks the field as dirty, even if the new value is identical to the old value. This special case is required to allow the user to mark an array-type field as dirty without having to call the `JDOHelper` method `makeDirty`.

5.5.4 Hollow (Required)

JDO instances that represent specific persistent data in the datastore but whose values are not in the JDO instance are hollow. The hollow state provides for the guarantee of uniqueness for persistent instances between transactions.

This is permitted to be the state of instances committed from a previous transaction, acquired by the method `getObjectById`, returned by iterating an `Extent`, returned in the result of a query execution, or navigating a persistent field reference. However, the JDO implementation may choose to return instances in a different state reachable from hollow.

A JDO implementation is permitted to effect a legal state transition of a hollow instance at any time, as if a field were read. Therefore, the hollow state might not be visible to the application.

During the commit of the transaction in which a dirty persistent instance has had its values changed (including a new persistent instance), the underlying datastore is changed to have the transactionally consistent values from the JDO instance, and the instance transitions to hollow.

Requests by the application for an instance with the same JDO identity (query, navigation, or lookup by `ObjectId`), in a subsequent transaction using the same `PersistenceManager` instance, will return the identical Java instance, assuming it has not been garbage collected. If the application does not hold a strong reference to a hollow instance, the instance might be garbage collected, as the `PersistenceManager` must not hold a strong reference to any hollow instance.

The hollow JDO instance maintains its JDO identity and its association with the JDO `PersistenceManager`. If the instance is of a class using application identity, the hollow instance maintains its primary key fields.

A hollow instance transitions to persistent-deleted if it is the parameter of `deletePersistent`.

A hollow instance transitions to persistent-dirty if a change is made to any managed field. It transitions to persistent-clean if a read access is made to any persistent field other than one of the primary key fields.

5.5.5 Persistent-clean (Required)

JDO instances that represent specific transactional persistent data in the datastore, and whose values have not been changed in the current transaction, are persistent-clean. This is the state of an instance whose values have been requested in the current datastore transaction, and whose values have not been changed by the current transaction.

A persistent-clean instance transitions to persistent-dirty if a change is made to any managed field.

A persistent-clean instance transitions to persistent-deleted if it is the parameter of `deletePersistent`.

A persistent-clean instance transitions to hollow at commit when `retainValues` is false; or rollback when `restoreValues` is false. It retains its identity and its association with the `PersistenceManager`.

5.5.6 Persistent-deleted (Required)

JDO instances that represent specific persistent data in the datastore, and that have been deleted in the current transaction, are persistent-deleted.

Read access to primary key fields is permitted but any other access to persistent fields will throw a `JDOUserException`.

Before the transition to persistent-deleted, the user-written `jdoPreDelete` is called if the persistence-capable class implements `InstanceCallbacks`.

A persistent-deleted instance transitions to transient at commit. During the transition, its persistent fields are written with their Java default values, and the instance loses its JDO Identity and its association with the `PersistenceManager`.

A persistent-deleted instance transitions to hollow at rollback when `restoreValues` is `false`. The instance retains its JDO Identity and its association with the `PersistenceManager`.

5.5.7 Persistent-new-deleted (Required)

JDO instances that represent instances that have been newly made persistent and deleted in the current transaction are persistent-new-deleted.

Read access to primary key fields is permitted but any other access to persistent fields will throw a `JDOUserException`.

Before the transition to persistent-new-deleted, the user-written `jdoPreDelete` is called if the persistence-capable class implements `InstanceCallbacks`.

A persistent-new-deleted instance transitions to transient at commit. During the transition, its persistent fields are written with their Java default values, and the instance loses its JDO Identity and its association with the `PersistenceManager`.

A persistent-new-deleted instance transitions to transient at rollback. The instance loses its JDO Identity and its association with the `PersistenceManager`.

If `RestoreValues` is `true`, the values of managed fields in the instance are restored to their state as of the call to `makePersistent`. If `RestoreValues` is `false`, the values of managed fields in the instance are left as they were at the time rollback was called.

5.6 Nontransactional (Optional)

Management of nontransactional instances is an optional feature of a JDO implementation. Usage is primarily for slowly changing data or for optimistic transaction management, as the values in nontransactional instances are not guaranteed to be transactionally consistent.

The use of this feature is governed by the `PersistenceManager` options `NontransactionalRead`, `NontransactionalWrite`, `Optimistic`, and `RetainValues`. An implementation might support any or all of these options. For example, an implementation might support only `NontransactionalRead`. For options that are not supported, the value of the unsupported property is `false` and it may not be changed.

If a `PersistenceManager` does not support this optional feature, an operation that would result in an instance transitioning to the persistent-nontransactional state or a request to set the `NontransactionalRead`, `NontransactionalWrite`, `Optimistic`, or `RetainValues` option to `true`, throws a `JDOUnsupportedOptionException`.

`NontransactionalRead`, `NontransactionalWrite`, `Optimistic`, and `RetainValues` are independent options. A JDO implementation must not automatically change the values of these properties as a side effect of the user changing other properties.

With `NontransactionalRead` set to `true`:

- Navigation and queries are valid outside a transaction. It is a JDO implementation decision whether the instances returned are in the hollow or persistent-nontransactional state.
- When a managed, non-key field of a hollow instance is read outside a transaction, the instance transitions to persistent-nontransactional.
- If a persistent-clean instance is the parameter of `makeNontransactional`, the instance transitions to persistent-nontransactional.

With `NontransactionalWrite` set to `true`:

- Modification of persistent-nontransactional instances is permitted outside a transaction. The changes do not participate in any subsequent transaction.

With `RetainValues` set to `true`:

- At commit, persistent-clean, persistent-new, and persistent-dirty instances transition to persistent-nontransactional. Fields defined in the XML metadata as containing mutable second-class types are examined to ensure that they contain instances that track changes made to them and are owned by the instance. If not, they are replaced with new second class object instances that track changes, constructed from the contents of the second class object instance. These include `java.util.Date`, and `Collection` and `Map` types.

NOTE: This process is not required to be recursive, although an implementation might choose to recursively convert the closure of the collection to become second class objects. JDO requires conversion only of the affected persistence-capable instance's fields.

With `RestoreValues` set to `true`:

- If the JDO implementation does not support persistent-nontransactional instances, at rollback persistent-deleted, persistent-clean and persistent-dirty instances transition to hollow.
- If the JDO implementation supports persistent-nontransactional instances, at rollback persistent-deleted, persistent-clean and persistent-dirty instances transition to persistent-nontransactional. The state of each managed field in persistent-deleted and persistent-dirty instances is restored:
 - fields of primitive types (`int`, `float`, etc.), wrapper types (`Integer`, `Float`, etc.), immutable types (`Locale`, etc.), and references to persistence-capable types are restored to their values as of the beginning of the transaction and the fields are marked as loaded.
 - fields of mutable types (`Date`, `Collection`, array-type, etc.) are set to `null` and the fields are marked as not loaded.

5.6.1 Persistent-nontransactional (Optional)

NOTE: The following discussion applies only to datastore transactions. See section 5.8 for a discussion on how optimistic transactions change this behavior.

JDO instances that represent specific persistent data in the datastore, whose values are currently loaded but not transactionally consistent, are persistent-nontransactional. There is a JDO Identity associated with these instances, and transactional instances can be obtained from the object ids.

The persistent-nontransactional state allows persistent instances to be managed as a shadow cache of instances that are updated asynchronously.

As long as a transaction is not in progress:

- if `NontransactionalRead` is `true`, persistent field values might be retrieved from the datastore by the `PersistenceManager`;
- if `NontransactionalWrite` is `true`, the application might make changes to the persistent field values in the instance, and
- There is no state change associated with either of the above operations.

A persistent-nontransactional instance transitions to persistent-clean if it is the parameter of a `makeTransactional` method executed when a transaction is in progress. The state of the instance in memory is discarded (cleared) and the state is loaded from the datastore.

A persistent-nontransactional instance transitions to persistent-clean if any managed field is accessed when a datastore transaction is in progress. The state of the instance in memory is discarded and the state is loaded from the datastore.

A persistent-nontransactional instance transitions to persistent-dirty if any managed field is written when a transaction is in progress. The state of the instance in memory is saved for use during rollback, and the state is loaded from the datastore. Then the change is applied.

A persistent-nontransactional instance transitions to persistent-deleted if it is the parameter of `deletePersistent`. The state of the instance in memory is saved for use during rollback.

If the application does not hold a strong reference to a persistent-nontransactional instance, the instance might be garbage collected. The `PersistenceManager` must not hold a strong reference to any persistent-nontransactional instance.

5.7 Transient Transactional (Optional)

Management of transient transactional instances is an optional feature of a JDO implementation. The following sections describe the additional states and state changes when using transient transactional behavior.

A transient instance transitions to transient-clean if it is the parameter of `makeTransactional`.

5.7.1 Transient-clean (Optional)

JDO instances that represent transient transactional instances whose values have not been changed in the current transaction are transient-clean. This state is not reachable if the JDO `PersistenceManager` does not implement `makeTransactional`.

Changes made outside a transaction are allowed without a state change. A transient-clean instance transitions to transient-dirty if any managed field is changed in a transaction. During the transition, values of managed fields are saved by the `PersistenceManager` for use during rollback.

A transient-clean instance transitions to transient if it is the parameter of `makeNon-transactional`.

5.7.2 Transient-dirty (Optional)

JDO instances that represent transient transactional instances whose values have been changed in the current transaction are transient-dirty. This state is not reachable if the JDO `PersistenceManager` does not implement `makeTransactional`.

A transient-dirty instance transitions to transient-clean at commit. The values of managed fields saved (for rollback processing) at the time the transition was made from transient-clean to transient-dirty are discarded. None of the values of fields in the instance are modified as a result of commit.

A transient-dirty instance transitions to transient-clean at rollback. The values of managed fields saved at the time the transition was made from transient-clean to transient-dirty are restored.

A transient-dirty instance transitions to persistent-new at `makePersistent`. The values of managed fields saved at the time the transition was made from transient-clean to transient-dirty are used as the before image for the purposes of rollback.

5.8 Optimistic Transactions (Optional)

Optimistic transaction management is an optional feature of a JDO implementation.

The `Optimistic` flag set to `true` changes the state transitions of persistent instances:

- If a persistent field other than one of the primary key fields is read, a hollow instance transitions to persistent-nontransactional instead of persistent-clean. Subsequent reads of these fields do not cause a transition from persistent-nontransactional.
- A persistent-nontransactional instance transitions to persistent-deleted if it is a parameter of `deletePersistent`. The state of the managed fields of the instance in memory is saved for use during rollback, and for verification during commit. The values in fields of the instance in memory are unchanged. If fresh values need to be loaded from the datastore, then the user should first call `refresh` on the instance.
- A persistent-nontransactional instance transitions to persistent-clean if it is a parameter of a `makeTransactional` method executed when an optimistic transaction is in progress. The values in managed fields of the instance in memory are unchanged. If fresh values need to be loaded from the datastore, then the user should first call `refresh` on the instance.
- A persistent-nontransactional instance transitions to persistent-dirty if a managed field is modified when an optimistic transaction is in progress. If `RestoreValues` is `true`, a before image is saved before the state transition. This is used for restoring field values during rollback. Depending on the implementation the before image of the instance in memory might be saved for verification during commit. The values in fields of the instance in memory are unchanged before the update is applied. If fresh values need to be loaded from the datastore, then the user should first call `refresh` on the instance.

Table 2: State Transitions

method \ current state	Transient	P-new	P-clean	P-dirty	Hollow
makePersistent	P-new	unchanged	unchanged	unchanged	unchanged
deletePersistent	error	P-new-del	P-del	P-del	P-del
makeTransactional	T-clean	unchanged	unchanged	unchanged	P-clean
makeNontransactional	error	error	P-nontrans	error	unchanged
makeTransient	unchanged	error	Transient	error	Transient
commit retainValues=false	unchanged	Hollow	Hollow	Hollow	unchanged
commit retainValues=true	unchanged	P-nontrans	P-nontrans	P-nontrans	unchanged
rollback restoreValues=false	unchanged	Transient	Hollow	Hollow	unchanged
rollback restoreValues=true	unchanged	Transient	P-nontrans	P-nontrans	unchanged
refresh with active Datastore transaction	unchanged	unchanged	unchanged	P-clean	unchanged
refresh with active Opti- mistic transaction	unchanged	unchanged	unchanged	P-nontrans	unchanged
evict	n/a	unchanged	Hollow	unchanged	unchanged
read field outside transac- tion	unchanged	impossible	impossible	impossible	P-nontrans
read field with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
read field with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	P-nontrans
write field or makeDirty with active transaction	unchanged	unchanged	P-dirty	unchanged	P-dirty

Table 2: State Transitions

method \ current state	Transient	P-new	P-clean	P-dirty	Hollow
retrieve outside or with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
retrieve with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean

method \ current state	T-clean	T-dirty	P-new-del	P-del	P-nontrans
makePersistent	P-new	P-new	unchanged	unchanged	unchanged
deletePersistent	error	error	unchanged	unchanged	P-del
makeTransactional	unchanged	unchanged	unchanged	unchanged	P-clean
makeNontransactional	Transient	error	error	error	unchanged
makeTransient	unchanged	unchanged	error	error	Transient
commit retainValues=false	unchanged	T-clean	Transient	Transient	unchanged
commit retainValues=true	unchanged	T-clean	Transient	Transient	unchanged
rollback restoreValues=false	unchanged	T-clean	Transient	Hollow	unchanged
rollback restoreValues=true	unchanged	T-clean	Transient	P-nontrans	unchanged
refresh	unchanged	unchanged	unchanged	unchanged	unchanged
evict	unchanged	unchanged	unchanged	unchanged	Hollow
read field outside transaction	unchanged	impossible	impossible	impossible	unchanged
read field with Optimistic transaction	unchanged	unchanged	error	error	unchanged
read field with active Datastore transaction	unchanged	unchanged	error	error	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	unchanged

method \ current state	T-clean	T-dirty	P-new-del	P-del	P-nontrans
write field or makeDirty with active transaction	T-dirty	unchanged	error	error	P-dirty
retrieve outside or with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	unchanged
retrieve with active Database transaction	unchanged	unchanged	unchanged	unchanged	P-clean

error: a `JDOUserException` is thrown; the state does not change

unchanged: no state change takes place; no exception is thrown due to the state change

n/a: not applicable; if this instance is an explicit parameter of the method, a `JDOUserException` is thrown; if this instance is an implicit parameter, it is ignored.

impossible: the state cannot occur in this scenario

Figure 7.0 Life Cycle: New Persistent Instances

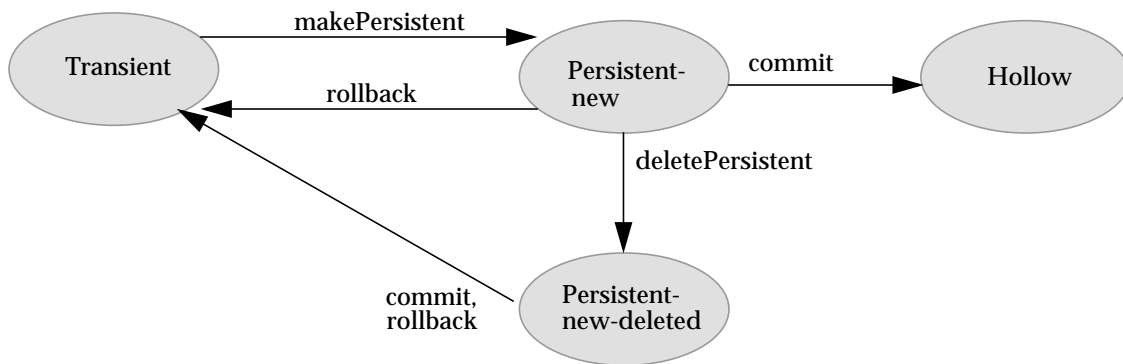


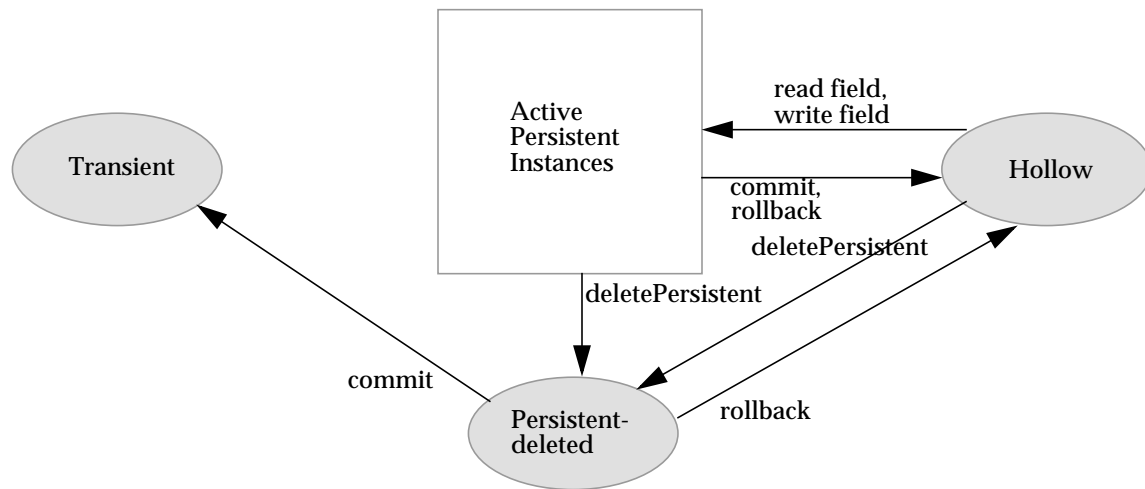
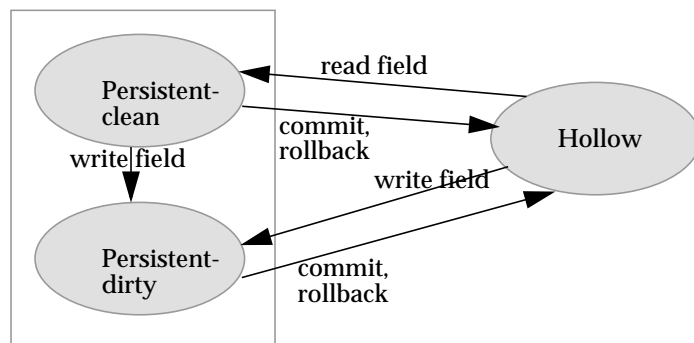
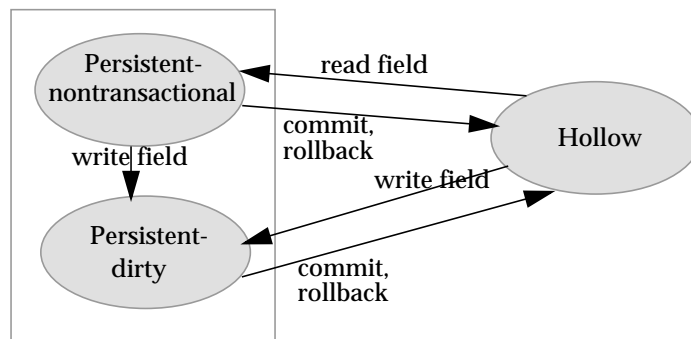
Figure 8.0 Life Cycle: Transactional Access**Figure 9.0** Life Cycle: Datastore Transactions**Figure 10.0** Life Cycle: Optimistic Transactions

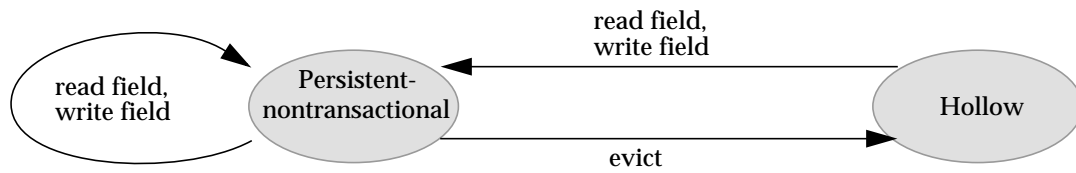
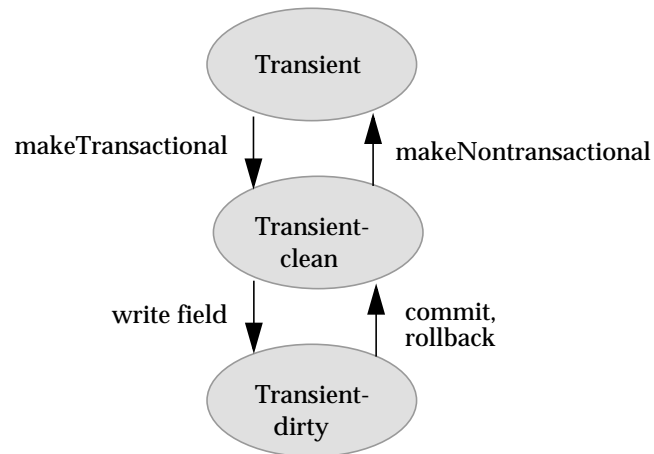
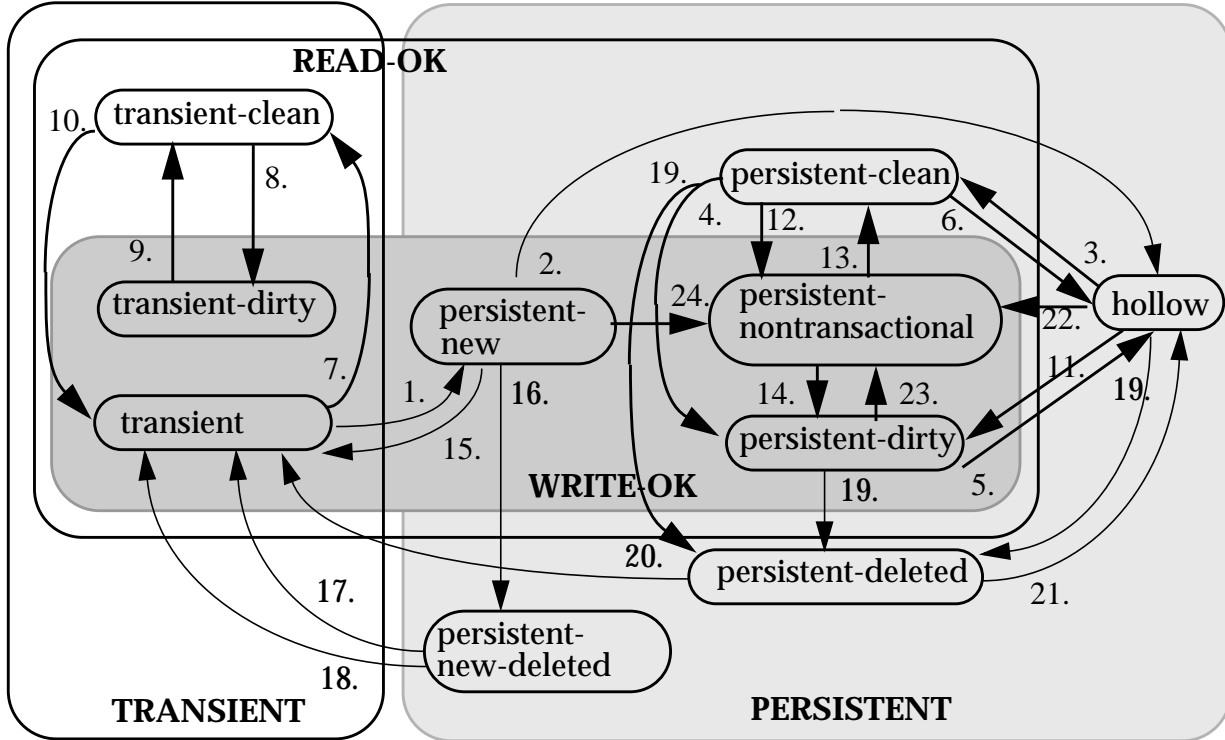
Figure 11.0 Life Cycle: Access Outside Transactions**Figure 12.0** Life Cycle: Transient Transactional

Figure 13.0 JDO Instance State Transitions

NOTE: Not all possible state transitions are shown in this diagram.

1. A transient instance transitions to persistent-new when the instance is the parameter of a `makePersistent` method.
2. A persistent-new instance transitions to hollow when the transaction in which it was made persistent commits.
3. A hollow instance transitions to persistent-clean when a field is read.
4. A persistent-clean instance transitions to persistent-dirty when a field is written.
5. A persistent-dirty instance transitions to hollow at commit or rollback.
6. A persistent-clean instance transitions to hollow at commit or rollback.
7. A transient instance transitions to transient-clean when it is the parameter of a `makeTransactional` method.
8. A transient-clean instance transitions to transient-dirty when a field is written.
9. A transient-dirty instance transitions to transient-clean at commit or rollback.
10. A transient-clean instance transitions to transient when it is the parameter of a `makeNontransactional` method.
11. A hollow instance transitions to persistent-dirty when a field is written.

12. A persistent-clean instance transitions to persistent-nontransactional at commit when `RetainValues` is set to `true`, at rollback when `RestoreValues` is set to `true`, or when it is the parameter of a `makeNontransactional` method.
13. A persistent-nontransactional instance transitions to persistent-clean when it is the parameter of a `makeTransactional` method.
14. A persistent-nontransactional instance transitions to persistent-dirty when a field is written in a transaction.
15. A persistent-new instance transitions to transient on rollback.
16. A persistent-new instance transitions to persistent-new-deleted when it is the parameter of `deletePersistent`.
17. A persistent-new-deleted instance transitions to transient on rollback. The values of the fields are restored as of the `makePersistent` method.
18. A persistent-new-deleted instance transitions to transient on commit. No changes are made to the values.
19. A hollow, persistent-clean, or persistent-dirty instance transitions to persistent-deleted when it is the parameter of `deletePersistent`.
20. A persistent-deleted instance transitions to transient when the transaction in which it was deleted commits.
21. A persistent-deleted instance transitions to hollow when the transaction in which it was deleted rolls back.
22. A hollow instance transitions to persistent-nontransactional when the `NontransactionalRead` option is set to `true`, a field is read, and there is either an optimistic transaction or no transaction active.
23. A persistent-dirty instance transitions to persistent-nontransactional at commit when `RetainValues` is set to `true` or at rollback when `RestoreValues` is set to `true`.
24. A persistent-new instance transitions to persistent-nontransactional at commit when `RetainValues` is set to `true`.

6 The Persistent Object Model

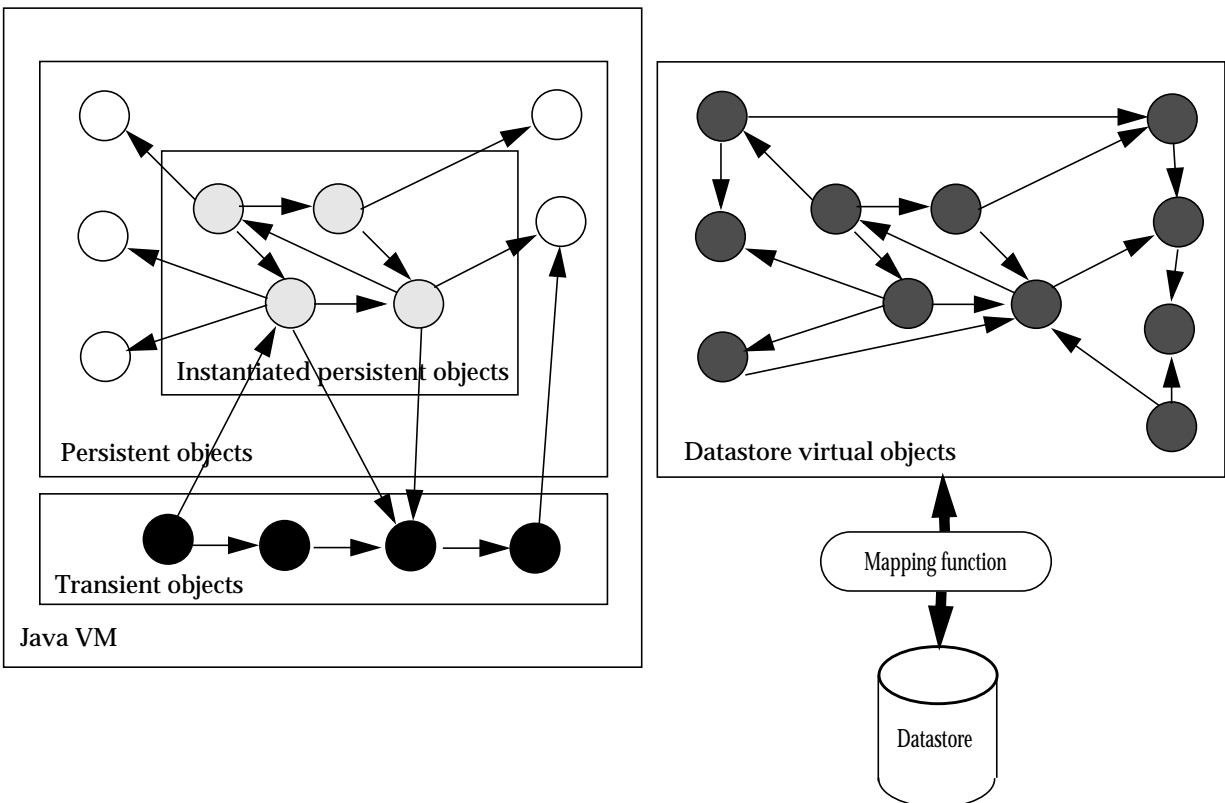
This chapter specifies the object model for persistence capable classes. To the extent possible, the object model is the same as the Java object model. Differences between the Java object model and the JDO object model are highlighted.

6.1 Overview

The Java execution environment supports different kinds of classes that are of interest to the developer. The classes that model the application and business domain are the primary focus of JDO. In a typical application, application classes are highly interconnected, and the graph of instances of those classes includes the entire contents of the datastore.

Applications typically deal with a small number of persistent instances at a time, and it is the function of JDO to allow the illusion that the application can access the entire graph of connected instances, while in reality only small subset of instances needs to be instantiated in the JVM. This concept is called transparent data access, transparent persistence, or simply transparency.

Figure 14.0 Instantiated persistent objects



Within a JVM, there may be multiple independent units of work that must be isolated from each other. This isolation imposes requirements on JDO to permit the instantiation of the same datastore object into multiple Java instances. The connected graph of Java instances is only a subset of the entire contents of the datastore. Whenever a reference is followed from one persistent instance to another, the JDO implementation transparently instantiates the required instance into the JVM.

The storage of objects in datastores might be quite different from the storage of objects in the JVM. Therefore, there is a mapping between the Java instances and the objects in the datastore. This mapping is performed by the JDO implementation, using metadata that is available at runtime. The metadata is generated by a JDO vendor-supplied tool, in cooperation with the deployer of the system. The mapping is not standardized by JDO.

JDO instances are stored in the datastore and retrieved, possibly field by field, from the datastore at specific points in their life cycle. The class developer might use callbacks at certain points to make a JDO instance ready for execution in the JVM, or make a JDO instance ready to be removed from the JVM. While executing in the JVM, a JDO instance might be connected to other instances, both persistent and transient.

There is no restriction on the types of non-persistent fields of persistence-capable classes. These fields behave exactly as defined by the Java language. Persistent fields of persistence-capable classes have restrictions in JDO, based on the characteristics of the types of the fields in the class definition.

6.2 Goals

The JDO Object Model has the following objectives:

- All field types supported by the Java language, including primitive types, reference types and interface types should be supported by JDO instances.
- All class and field modifiers supported by the Java language including private, public, protected, static, transient, abstract, final, synchronized, and volatile, should be supported by JDO instances.
- All user-defined classes should be allowed to be persistence-capable.
- Some system-defined classes (especially those for modeling state) should be persistence-capable.

6.3 Architecture

In Java, variables (including fields of classes) have types. Types are either primitive types or reference types. Reference types are either classes or interfaces. Arrays are treated as classes.

An object is an instance of a specific class, determined when the instance is constructed. Instances may be assigned to variables if they are assignment compatible with the variable type.

PersistenceCapable interface

The JDO Object Model distinguishes between two kinds of classes: those that implement `PersistenceCapable` and those that don't. A user-defined class can implement `PersistenceCapable` unless its state depends on the state of inaccessible or remote objects (e.g. it extends `java.net.SocketImpl` or uses JNI (native calls) to implement `ja-`

`va.net.SocketOptions`). A non-static inner class cannot be persistence-capable because the state of its instances depends on the state of their enclosing instances.

Except for system-defined classes specially addressed by the JDO specification, system-defined classes (those defined in `java.lang`, `java.io`, `java.util`, `java.net`, etc.) are not persistence-capable, nor is a system-defined class allowed to be the type of a persistent field.

First Class Objects and Second Class Objects

A First Class Object (FCO) is an instance of `PersistenceCapable` that has JDO Identity and can be stored in a datastore, and independently deleted and queried. A Second Class Object (SCO) has no JDO Identity of its own and is stored in the datastore only as part of a First Class Object. In some JDO implementations, some SCO instances are actually artifacts that have no literal datastore representation at all, but are used only to represent relationships. For example, a `Collection` of a `PersistenceCapable` class might not be stored in the datastore, but created when needed to represent the relationship in memory. At commit time, the memory artifact is discarded and the relationship is represented entirely by datastore relationships.

First Class Objects

FCOs support uniquing; whenever an FCO is instantiated into memory, there is guaranteed to be only one instance representing that FCO managed by the same `PersistenceManager` instance. They are passed as arguments by reference.

An FCO can be shared among multiple FCOs, and if an FCO is changed (and the change is committed to the datastore), then the changes are visible to all other FCOs that refer to it.

Second Class Objects

Second Class Objects are either instances of immutable system classes (`java.lang.Integer`, `java.lang.String`, etc.), JDO implementation subclasses of mutable system classes that implement the functionality of their system class (`java.util.Date`, `java.util.HashSet`, etc.), or `PersistenceCapable` classes.

Second Class Objects of mutable system classes and persistence-capable classes track changes made to them, and notify their owning FCO that they have changed. The change is reflected as a change to the owning FCO (e.g. the owning instance might change state from persistent-clean to persistent-dirty). They are stored in the datastore only as part of a FCO. They do not support uniquing, and the Java object identity of the values of the persistent fields containing them is lost when the owning FCO is flushed to the datastore. They are passed as arguments by reference.

SCO fields must be explicitly or by default identified in the metadata as embedded. If a field, or an element of a collection or a map key or value is identified as embedded (embedded-element, embedded-key, or embedded-value) then any instances so identified in the collection or map are treated as SCO during commit. That is, the value is stored with the owning FCO and the value loses its own identity if it had one.

SCO fields of persistence-capable types are identified as embedded. The behavior of embedded persistence-capable types is intended to mirror the behavior of system types, but this is not standard, and portable applications must not depend on this behavior.

It is possible for an application to assign the same instance of a mutable SCO class to multiple FCO embedded fields, but this non-portable behavior is strongly discouraged for the following reason. If the assignment is done to persistent-new, persistent-clean, or persistent-dirty instances, then at the time that the FCOs are committed to the datastore, the Java object identity of the owned SCOs might change, because each FCO might have its own

unshared SCO. If the assignment is done before `makePersistent` is called to make the FCOs persistent, the embedded fields are immediately replaced by copies, and no sharing takes place.

When an FCO is instantiated in the JVM by a JDO implementation, and an embedded field of a mutable type is accessed, the JDO implementation assigns to these fields a new instance that tracks changes made to itself, and notifies the owning FCO of the change. Similarly, when an FCO is made persistent, either by being the parameter of `makePersistent` or `makePersistentAll` or by being reachable from a parameter of `makePersistent` or `makePersistentAll` at the time of the execution of the `makePersistent` or `makePersistentAll` method call, the JDO implementation replaces the field values of mutable SCO types with instances of JDO implementation subclasses of the mutable system types.

Therefore, the application cannot assume that it knows the actual class of instances assigned to SCO fields, although it is guaranteed that the actual class is assignment compatible with the type.

There are few differences visible to the application between a field mapped to an FCO and an SCO. One difference is in sharing. If an FCO1 is assigned to a persistent field in FCO2 and FCO3, then any changes at any time to instance FCO1 will be visible from FCO2 and FCO3.

If an SCO1 is assigned to a persistent field in persistent instances FCO1 and FCO2, then any changes to SCO1 will be visible from instances FCO1 and FCO2 only until FCO1 and FCO2 are committed. After commit, instance SCO1 might not be referenced by either FCO1 or FCO2, and any changes made to SCO1 might not be reflected in either FCO1 or FCO2.

Another difference is in visibility of SCO instances by queries. SCO instances are not added to `Extents`. If the SCO instance is of a `PersistenceCapable` type, it is not visible to queries of the `Extent` of the `PersistenceCapable`. Furthermore, the field values of SCO instances of `PersistenceCapable` types might not be visible to queries at all.

Sharing of immutable SCO fields is supported in that it is good practice to assign the same immutable instance to multiple SCO fields. But the field values should not be compared using Java identity, but only by Java equality. This is the same good practice used with non-persistent instances.

Arrays

Arrays are system-defined classes that do not necessarily have any JDO Identity of their own, and support by a JDO implementation is optional. If an implementation supports them, they might be stored in the datastore as part of an FCO. They do not support uniqueness, and the Java object identity of the values of the persistent fields containing them is lost when the owning FCO is flushed to the datastore. They are passed as arguments by reference.

Tracking changes to Arrays is not required to be done by a JDO implementation. If an Array owned by an FCO is changed, then the changes might not be flushed to the datastore. Portable applications must not require that these changes be tracked. In order for changes to arrays to be tracked, the application must explicitly notify the owning FCO of the change to the Array by calling the `jdoMakeDirty` method of the `PersistenceCapable` interface (or `makeDirty` of the `JDOHelper` class), or by replacing the field value with its current value.

Since changes to array-type fields cannot be tracked by JDO, setting the value of an array-type managed field marks the field as dirty, even if the new value is identical to the old

value. This special case is required to allow the user to mark an array-type field as dirty without having to call the JDOHelper method `makeDirty`.

Furthermore, an implementation is permitted, but not required to, track changes to Arrays passed as references outside the body of methods of the owning class. There is a method defined on interface `PersistenceCapable` that allows the application to mark the field containing such an Array to be modified so its changes can be tracked. Portable applications must not require that these changes be tracked automatically. When a reference to the Array is returned as a result of a method call, a portable application first marks the Array field as dirty.

It is possible for an application to assign the same instance of an Array to multiple FCOs, but after the FCO is flushed to the datastore, the Java object identity of the Array might change.

When an FCO is instantiated in the JVM, the JDO implementation assigns to fields with an Array type a new instance with a different Java object identity from the instance stored.

Therefore, the application cannot assume that it knows the identity of instances assigned to Array fields, although it is guaranteed that the actual value is the same as the value stored.

Primitives

Primitives are types defined in the Java language and comprise `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`. They might be stored in the datastore only as part of an FCO. They have no Java identity and no datastore identity of their own. They are passed as arguments by value.

Interfaces

Interfaces are types whose values may be instances of any class that declare that they implement that interface.

6.4 Field types of persistence-capable classes

6.4.1 Nontransactional non-persistent fields

There are no restrictions on the types of nontransactional non-persistent fields. These fields are managed entirely by the application, not by the JDO implementation. Their state is not preserved by the JDO implementation, although they might be modified during execution of user-written callbacks defined in interface `InstanceCallbacks` at specific points in the life cycle, or any time during the instance's existence in the JVM.

6.4.2 Transactional non-persistent fields

There are no restrictions on the types of transactional non-persistent fields. These fields are partly managed by the JDO implementation. Their state is preserved and restored by the JDO implementation during certain state transitions.

6.4.3 Persistent fields

Primitive types

JDO implementations must support fields of any of the primitive types

- `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

Primitive values are stored in the datastore associated with their owning FCO. They have no JDO Identity.

Immutable Object Class types

JDO implementations must support fields that reference instances of immutable object classes, and may choose to support these instances as SCOs or FCOs:

- package `java.lang`: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `String`;
- package `java.util`: `Locale`;
- package `java.math`: `BigDecimal`, `BigInteger`.

Portable JDO applications must not depend on whether instances of these classes are treated as SCOs or FCOs.

Mutable Object Class types

JDO implementations must support fields that reference instances of the following mutable object classes, and may choose to support these instances as SCOs or FCOs:

- package `java.util`: `Date`, `HashSet`.

JDO implementations may optionally support fields that reference instances of the following mutable object classes, and may choose to support these instances as SCOs or FCOs:

- package `java.util`: `ArrayList`, `HashMap`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, and `Vector`.

Because the treatment of these fields may be as SCO, the behavior of these mutable object classes when used in a persistent instance is not identical to their behavior in a transient instance.

Portable JDO applications must not depend on whether instances of these classes referenced by fields are treated as SCOs or FCOs.

PersistenceCapable Class types

JDO implementations must support references to FCO instances of `PersistenceCapable` and are permitted, but not required, to support references to SCO instances of `PersistenceCapable`.

Portable JDO applications must not depend on whether these fields are treated as SCOs or FCOs.

Object Class type

JDO implementations must support fields of `Object` class type as FCOs. The implementation is permitted, but is not required, to allow any class to be assigned to the field. If an implementation restricts instances to be assigned to the field, a `ClassCastException` must be thrown at the time of any incorrect assignment.

Portable JDO applications must not depend on whether these fields are treated as SCOs or FCOs.

Collection Interface types

JDO implementations must support fields of interface types, and may choose to support them as SCOs or FCOs: package `java.util`: `Collection`, `Map`, `Set`, and `List`. `Collection` and `Set` are required; `Map` and `List` are optional.

Portable JDO applications must not depend on whether these fields are treated as SCOs or FCOs.

Other Interface types

JDO implementations must support fields of interface types other than `Collection` interface types as FCOs. The implementation is permitted, but is not required, to allow any class that implements the interface to be assigned to the field. If an implementation further restricts instances that can be assigned to the field, a `ClassCastException` must be thrown at the time of any incorrect assignment.

Portable JDO applications must treat these fields as FCOs.

Arrays

JDO implementations may optionally support fields of array types, and may choose to support them as SCOs or FCOs. If Arrays are supported by JDO implementations, they are permitted, but not required, to track changes made to Arrays that are fields of persistence capable classes in the methods of the classes. They need not track changes made to Arrays that are passed by reference as arguments to methods, including methods of persistence-capable classes.

Portable JDO applications must not depend on whether these fields are treated as SCOs or FCOs.

6.5 Inheritance

A class might be persistence-capable even if its superclass is not persistence-capable. This allows users to extend classes that were not designed to be persistence-capable. If a class is persistence-capable, then its subclasses might or might not be persistence-capable themselves.

Further, subclasses of such classes that are not persistence-capable might be persistence-capable. That is, it is possible for classes in the inheritance hierarchy to be independently persistence-capable and not persistence-capable. It is not sufficient to test if a class implements `PersistenceCapable` (e.g. testing `anInstance instanceof PersistenceCapable`) to determine whether an instance is allowed to be stored.

Fields identified in the XML metadata as persistent or transactional in persistence-capable classes must be fields declared in that Java class definition. That is, inherited fields cannot be named in the XML metadata.

Fields identified as persistent in persistence-capable classes will be persistent in subclasses; fields identified as transactional in persistence-capable classes will be transactional in subclasses; and fields identified as non-persistent in persistence-capable classes will be non-persistent in subclasses.

Of course, a class might define a new field with the same name as the field declared in the superclass, and might define it with a different persistence-modifier from the inherited field. But Java treats the declared field as a different field from the inherited field, so there is no conflict.

All persistence-capable classes must have a no-arg constructor. This constructor might be a private constructor, as it is only used from within the `jdoNewInstance` methods. The constructor might be the default no-arg constructor created by the compiler when the source code does not define any constructors.

The identity type of the least-derived persistence-capable class defines the identity type for all persistence-capable classes that extend it.

Persistence-capable classes that use application identity have special considerations for inheritance:

Key fields may be declared only in abstract superclasses and least-derived concrete classes in inheritance hierarchies. Key fields declared in these classes must also be declared in the corresponding objectid classes, and the objectid classes must form an inheritance hierarchy corresponding to the inheritance hierarchy of the persistence-capable classes. A persistence-capable class can only have one concrete objectid class anywhere in its inheritance hierarchy.

For example, if an abstract class `Component` declares a key field `masterId`, the objectid class `ComponentKey` must also declare a field of the same type and name. If `ComponentKey` is concrete, then no subclass is allowed to define an objectid class.

If `ComponentKey` is abstract, an instance of a concrete subclass of `ComponentKey` must be used to find a persistent instance. A concrete class `Part` that extends `Component` must declare a concrete objectid class (for example, `PartKey`) that extends `ComponentKey`. There might be no key fields declared in `Part` or `PartKey`. Persistence-capable subclasses of `Part` must not have an objectid class.

Another concrete class `Assembly` that extends `Component` must declare a concrete objectid class (for example, `AssemblyKey`) that extends `ComponentKey`. If there is a key field, it must be declared in both `Assembly` and `AssemblyKey`. Persistence-capable subclasses of `Assembly` must not have an objectid class.

There might be other abstract classes or non-persistence-capable classes in the inheritance hierarchy between `Component` and `Part`, or between `Component` and `Assembly`. These classes are ignored for the purposes of objectid classes and key fields.

7 PersistenceCapable

Every instance that is managed by a JDO `PersistenceManager` must be of a class that implements the public `PersistenceCapable` interface. This interface defines methods that allow the implementation to manage the instances. It also defines methods that allow a JDO aware application to examine the runtime state of instances, for example to discover whether the instance is transient, persistent, transactional, dirty, etc., and to discover its associated `PersistenceManager` if it has one.

The JDO Reference Enhancer modifies the class to implement `PersistenceCapable` prior to loading the class into the runtime environment. The enhancer additionally adds code to implement the methods defined by `PersistenceCapable`.

The `PersistenceCapable` interface is designed to avoid name conflicts in the scope of user-defined classes. All of its declared method names are prefixed with “jdo”.

Class implementors may explicitly declare that the class implements `PersistenceCapable`. If this is done, the implementor must implement the `PersistenceCapable` contract, and the enhancer will ignore the class instead of enhancing it.

The recommended approach for applications to interrogate the state of the instance is to use the class `JDOHelper`, which provides static methods that delegate to the instance if it implements `PersistenceCapable`, and if not, returns the values that would have been returned by a transient instance.

NOTE: This interface is not intended to be used by application programmers. It is for use only by implementations. Applications should use the methods defined in class `JDOHelper` instead of these methods.

```
package javax.jdo.spi;
interface PersistenceCapable {
```

7.1 Persistence Manager

```
PersistenceManager jdoGetPersistenceManager();
```

This method returns the associated `PersistenceManager` or null if the instance is transient.

7.2 Make Dirty

```
void jdoMakeDirty (String fieldName);
```

This method marks the specified field dirty so that its values will be modified in the data-store when the transaction in which the instance is modified is committed. The `fieldName` is the name of the field to be marked as dirty, optionally including the fully qualified package name and class name of the field. This method returns with no effect if the in-

stance is not managed by a `StateManager`. This method has the same effect on the life cycle state of the instance as changing a managed field would.

If the same name is used for multiple fields (a class declares a field of the same name as a field in one of its superclasses) then the unqualified name refers to the most-derived class in which the field is declared to be persistent. The qualified name (`className.fieldName`) should always be used to identify the field to avoid ambiguity with subclass-defined fields.

The rationale for this is that a method in a superclass might call this method, and specify the name of the field that is hidden by a subclass. The `StateManager` has no way of knowing which class called this method, and therefore assumes the Java rule regarding field names.

It is always safe to explicitly name the class and field referred to in the parameter to the method. The `StateManager` will resolve the scope of the name in the class named in the parameter.

For example, if class C inherits class B which inherits class A, and field X is declared in classes A and C, a method declared in class B may refer to the field in the method as “B.X” and it will refer to the field declared in class A. Field X is not declared in B; however, in the scope of class B, X refers to A.X.

7.3 JDO Identity

```
Object jdoGetObjectId();
```

This method returns the JDO identity of the instance. If the instance is transient, `null` is returned. If the identity is being changed in a transaction, this method returns the identity as of the beginning of the transaction.

```
Object jdoGetTransactionalObjectId();
```

This method returns the JDO identity of the instance. If the instance is transient, `null` is returned. If the identity is being changed in a transaction, this method returns the current identity in the transaction.

7.4 Status interrogation

The status interrogation methods return a boolean that represents the state of the instance:

7.4.1 Dirty

```
boolean jdoIsDirty();
```

Instances whose state has been changed in the current transaction return `true`. If the instance is transient, `false` is returned.

7.4.2 Transactional

```
boolean jdoIsTransactional();
```

Instances whose state is associated with the current transaction return `true`. If the instance is transient, `false` is returned.

7.4.3 Persistent

```
boolean jdoIsPersistent();
```

Instances that represent persistent objects in the datastore return `true`. If the instance is transient, `false` is returned.

7.4.4 New

```
boolean jdoIsNew();
```

Instances that have been made persistent in the current transaction return `true`. If the instance is transient, `false` is returned.

7.4.5 Deleted

```
boolean jdoIsDeleted();
```

Instances that have been deleted in the current transaction return `true`. If the instance is transient, `false` is returned.

Table 3: State interrogation

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

7.5 New instance

```
PersistenceCapable jdoNewInstance(StateManager sm);
```

This method creates a new instance of the class of the instance. It is intended to be used as a performance optimization compared to constructing a new instance by reflection using the constructor. It is intended to be used only by JDO implementations, not by applications. If the class is abstract, `null` is returned.

```
PersistenceCapable jdoNewInstance(StateManager sm, Object oid);
```

This method creates a new instance of the class of the instance, and copies key field values from the `oid` parameter instance. It is intended to be used as a performance optimization compared to constructing a new instance by reflection using the constructor, and copying

values from the oid instance by reflection. It is intended to be used only by JDO implementations for classes that use application identity, not by applications. If the class is abstract, null is returned.

7.6 State Manager

```
void jdoReplaceStateManager (StateManager sm)
    throws SecurityException;
```

This method sets the `jdoStateManager` field to the parameter. This method is normally used by the `StateManager` during the process of making an instance persistent, transactional, or transient. The caller of this method must have `JDOPermission("set-StateManager")` for the instance, otherwise `SecurityException` is thrown.

7.7 Replace Flags

```
void jdoReplaceFlags ();
```

This method tells the instance to call the owning `StateManager`'s `replacingFlags` method to get a new value for the `jdoFlags` field.

7.8 Replace Fields

```
void jdoReplaceField (int fieldNumber);
```

This method gets a new value from the `StateManager` for the field specified in the parameter. The field number must refer to a field declared in this class or in a superclass.

```
void jdoReplaceFields (int[] fieldNumbers);
```

This method iterates over the array of field numbers and calls `jdoReplaceField` for each one.

7.9 Provide Fields

```
void jdoProvideField (int fieldNumber);
```

This method provides the value of the specified field to the `StateManager`. The field number must refer to a field declared in this class or in a superclass.

```
void jdoProvideFields (int[] fieldNumbers);
```

This method iterates over the array of field numbers and calls `jdoProvideField` for each one.

7.10 Copy Fields

```
void jdoCopyFields (Object other, int[] fieldNumbers);
```

This method copies fields from another instance of the same class. This method can be invoked only when both `this` and `other` are managed by the same `StateManager`.

7.11 Static Fields

The following fields define the permitted values for the `jdoFlags` field.

```
public static final byte READ_WRITE_OK = 0;
public static final byte READ_OK = -1;
public static final byte LOAD_REQUIRED = 1;
```

The following fields define the flags for the `jdoFieldFlags` elements.

```
public static final byte CHECK_READ = 1;
public static final byte MEDIATE_READ = 2;
public static final byte CHECK_WRITE = 4;
public static final byte MEDIATE_WRITE = 8;
public static final byte SERIALIZABLE = 16;
```

7.12 JDO identity handling

```
public Object jdoNewObjectIdInstance();
```

This method creates a new instance of the class used for JDO identity. It is intended only for application identity. If the class has been enhanced for datastore identity, or if the class is abstract, `null` is returned.

```
public Object jdoNewObjectIdInstance(String str);
```

This method creates a new instance of the class used for JDO identity, using the `String` constructor of the object id class. It is intended only for application identity. If the class has been enhanced for datastore identity, or if the class is abstract, `null` is returned.

```
public void jdoCopyKeyFieldsToObjectId(Object oid);
```

This method copies all key fields from this instance to the parameter. The first parameter be an instance of the JDO identity class, or `ClassCastException` is thrown.

```
public void jdoCopyKeyFieldsToObjectId(ObjectIdFieldSupplier
fs, Object oid);
```

This method copies fields from the field manager instance to the second parameter instance. Each key field in the `ObjectId` class matching a key field in the `PersistenceCapable` class is set by the execution of this method. For each key field, the method of the `ObjectIdFieldSupplier` is called for the corresponding type of field. The second parameter must be an instance of the JDO identity class. If the parameter is not of the correct type, then `ClassCastException` is thrown.

```
public void jdoCopyKeyFieldsFromObjectId(ObjectIdFieldConsumer
fc, Object oid);
```

This method copies fields to the field manager instance from the second parameter instance. Each key field in the `ObjectId` class matching a key field in the `PersistenceCapable` class is retrieved by the execution of this method. For each key field, the method of the `ObjectIdFieldConsumer` is called for the corresponding type of field. The second parameter must be an instance of the JDO identity class. If the parameter is not of the correct type, then `ClassCastException` is thrown.

interface ObjectIdFieldSupplier

```
boolean fetchBooleanField (int fieldNumber);
char fetchCharField (int fieldNumber);
short fetchShortField (int fieldNumber);
```

```
int fetchIntField (int fieldNumber);
long fetchLongField (int fieldNumber);
float fetchFloatField (int fieldNumber);
double fetchDoubleField (int fieldNumber);
String fetchStringField (int fieldNumber);
Object fetchObjectField (int fieldNumber);
```

These methods all fetch one field from the field manager. The returned value is stored in the object id instance. The generated code in the `PersistenceCapable` class calls a method in the field manager for each key field in the object id. The field number is the same as in the persistence capable class for the corresponding key field.

interface ObjectIdFieldConsumer

```
void storeBooleanField (int fieldNumber, boolean value);
void storeCharField (int fieldNumber, char value);
void storeShortField (int fieldNumber, short value);
void storeIntField (int fieldNumber, int value);
void storeLongField (int fieldNumber, long value);
void storeFloatField (int fieldNumber, float value);
void storeDoubleField (int fieldNumber, double value);
void storeStringField (int fieldNumber, String value);
void storeObjectField (int fieldNumber, Object value);
```

These methods all store one field to the field manager. The value is retrieved from the object id instance. The generated code in the `PersistenceCapable` class calls a method in the field manager for each key field in the object id. The field number is the same as in the persistence capable class for the corresponding key field.

**interface ObjectIdFieldManager extends ObjectIdFieldSupplier,
 ObjectIdFieldConsumer**

This interface is a convenience interface that extends both `ObjectIdFieldSupplier` and `ObjectIdFieldConsumer`.

Readers primarily interested in developing applications with the JDO API can ignore the following chapters. Skip to 10 – InstanceCallbacks.

8 JDOHelper

JDOHelper is a class with static methods that is intended for use by persistence-aware classes. It contains methods that allow interrogation of the persistent state of an instance of a persistence-capable class.

Each method delegates to the instance, if it implements `PersistenceCapable`. Otherwise, if the method returns a value of reference type, it returns `null`; if the method returns a value of boolean type, it returns `false`; and if the method returns `void`, there is no effect.

```
package javax.jdo;
class JDOHelper {
```

8.1 Persistence Manager

```
static PersistenceManager getPersistenceManager (Object pc);
```

This method returns the associated `PersistenceManager`. It returns `null` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoGetPersistenceManager()`.

8.2 Make Dirty

```
static void makeDirty (Object pc, String fieldName);
```

This method marks the specified field dirty so that its values will be modified in the data-store when the instance is flushed. The `fieldName` is the name of the field to be marked as dirty, optionally including the fully qualified package name and class name of the field. This method has no effect if the instance is transient or `null`, or does not implement `PersistenceCapable`; or `fieldName` is not a managed field.

See also `PersistenceCapable.jdoMakeDirty(String fieldName)`.

8.3 JDO Identity

```
static Object getObjectId (Object pc);
```

This method returns the JDO identity of the instance. It returns `null` if the instance is transient or `null` or does not implement `PersistenceCapable`. If the identity is being changed in a transaction, this method returns the identity as of the beginning of the transaction.

See also `PersistenceCapable.jdoGetObjectId()` and `PersistenceManager.getObjectId(Object pc)`.

```
static Object getTransactionalObjectId (Object pc);
```

This method returns the JDO identity of the instance. It returns `null` if the instance is transient or `null` or does not implement `PersistenceCapable`. If the identity is being changed in a transaction, this method returns the current identity in the transaction.

See also `PersistenceCapable.jdoGetTransactionalObjectId()` and `PersistenceManager.getTransactionalObjectId(Object pc)`.

8.4 Status interrogation

The status interrogation methods return a boolean that represents the state of the instance:

8.4.1 Dirty

```
static boolean isDirty (Object pc);
```

Instances whose state has been changed in the current transaction return `true`. It returns `false` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoIsDirty()`;

8.4.2 Transactional

```
static boolean isTransactional (Object pc);
```

Instances whose state is associated with the current transaction return `true`. It returns `false` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoIsTransactional()`.

8.4.3 Persistent

```
static boolean isPersistent (Object pc);
```

Instances that represent persistent objects in the datastore return `true`. It returns `false` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoIsPersistent()`;

8.4.4 New

```
static boolean isNew (Object pc);
```

Instances that have been made persistent in the current transaction return `true`. It returns `false` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoIsNew()`;

8.4.5 Deleted

```
static boolean isDeleted (Object pc);
```

Instances that have been deleted in the current transaction return `true`. It returns `false` if the instance is transient or `null` or does not implement `PersistenceCapable`.

See also `PersistenceCapable.jdoIsDeleted()`;

8.5 PersistenceManagerFactory methods

```
public static
```

```
PersistenceManagerFactory getPersistenceManagerFactory  
    (Properties props, ClassLoader cl);  
public static  
    PersistenceManagerFactory getPersistenceManagerFactory  
        (Properties props);
```

These methods return a `PersistenceManagerFactory` based on properties contained in the `Properties` parameter. In the method without a class loader parameter, the calling thread's current `contextClassLoader` is used to resolve the class name.

This method delegates to the static method `getPersistenceManagerFactory` in the class named in the property `javax.jdo.PersistenceManagerFactoryClass`. If there are any exceptions while trying to call the static method, then either `JDOFatalUserException` or `JDOFatalInternalException` is thrown, depending on whether the exception is due to the user or the implementation. The nested exception indicates the cause of the exception.

If the class named by the `javax.jdo.PersistenceManagerFactoryClass` property cannot be found, or is not accessible to the user, then `JDOFatalUserException` is thrown. If there is no public static implementation of the `getPersistenceManagerFactory(Properties)` method, then `JDOFatalInternalException` is thrown. If the implementation of the static `getPersistenceManagerFactory(Properties)` method throws an exception, it is rethrown by this method.

The following are standard key values for the `Properties`:

```
javax.jdo.PersistenceManagerFactoryClass  
javax.jdo.option.Optimistic  
javax.jdo.option.RetainValues  
javax.jdo.option.RestoreValues  
javax.jdo.option.IgnoreCache  
javax.jdo.option.NontransactionalRead  
javax.jdo.option.NontransactionalWrite  
javax.jdo.option.Multithreaded  
javax.jdo.option.ConnectionDriverName  
javax.jdo.option.ConnectionUserName  
javax.jdo.option.ConnectionPassword  
javax.jdo.option.ConnectionURL  
javax.jdo.option.ConnectionFactoryName  
javax.jdo.option.ConnectionFactory2Name
```

JDO implementations are permitted to define key values of their own. Any key values not recognized by the implementation must be ignored. Key values that are recognized but not supported by an implementation must result in a `JDOFatalUserException` thrown by the method.

The returned `PersistenceManagerFactory` is not configurable (the `setXXX` methods will throw an exception). JDO implementations might manage a map of instantiated `PersistenceManagerFactory` instances based on specified property key values, and

return a previously instantiated `PersistenceManagerFactory` instance. In this case, the properties of the returned instance must exactly match the requested properties.

9 JDOImplHelper

This class is a public helper class for use by JDO implementations. It contains a registry of metadata by class. Use of the methods in this class avoids the use of reflection at runtime. `PersistenceCapable` classes register metadata with this class during class initialization.

NOTE: This interface is not intended to be used by application programmers. It is for use only by implementations.

```
package javax.jdo.spi;  
public JDOImplHelper {
```

9.1 JDOImplHelper access

```
public static JDOImplHelper getInstance()  
    throws SecurityException;
```

This method returns an instance of the `JDOImplHelper` class if the caller is authorized for `JDOPermission("getMetadata")`, and throws `SecurityException` if not authorized. This instance gives access to all of the other methods, except for `registerClass`, which is static and does not need any authorization.

9.2 Metadata access

```
public String[] getFieldNames (Class pcClass);
```

This method returns the names of persistent and transactional fields of the parameter class. If the class does not implement `PersistenceCapable`, or if it has not been enhanced correctly to register its metadata, a `JDOFatalUserException` is thrown.

Otherwise, the names of fields that are either persistent or transactional are returned, in order. The order of names in the returned array are the same as the field numbering. Relative field 0 refers to the first field in the array. The length of the array is the number of persistent and transactional fields in the class.

```
public Class[] getFieldTypes (Class pcClass);
```

This method returns the types of persistent and transactional fields of the parameter class. If the parameter does not implement `PersistenceCapable`, or if it has not been enhanced correctly to register its metadata, a `JDOFatalUserException` is thrown.

Otherwise, the types of fields that are either persistent or transactional are returned, in order. The order of types in the returned array is the same as the field numbering. Relative field 0 refers to the first field in the array. The length of the array is the number of persistent and transactional fields in the class.

```
public byte[] getFieldFlags (Class pcClass);
```


This method returns the field flags of persistent and transactional fields of the parameter class. If the parameter does not implement `PersistenceCapable`, or if it has not been enhanced correctly to register its metadata, a `JDOFatalUserException` is thrown.

Otherwise, the types of fields that are either persistent or transactional are returned, in order. The order of types in the returned array is the same as the field numbering. Relative field 0 refers to the first field in the array. The length of the array is the number of persistent and transactional fields in the class.

```
public Class getPersistenceCapableSuperclass (Class pcClass);
```

This method returns the `PersistenceCapable` superclass of the parameter class, or null if there is none.

9.3 Persistence-capable instance factory

```
public PersistenceCapable newInstance (Class pcClass,  
StateManager sm);
```

```
public PersistenceCapable newInstance (Class pcClass, StateMan-  
ager sm, Object oid);
```

If the class does not implement `PersistenceCapable`, or if it has not been enhanced correctly to register its metadata, a `JDOFatalUserException` is thrown. If the class is abstract, a `JDOFatalInternalException` is thrown.

Otherwise, a new instance of the class is constructed and initialized with the parameter `StateManager`. The new instance has its `jdoFlags` set to `LOAD_REQUIRED` but has no defined state. The behavior of the instance is determined by the owning `StateManager`.

The second form of the method returns a new instance of `PersistenceCapable` that has had its key fields initialized by the `ObjectId` parameter instance. If the class has been enhanced for datastore identity, then the `oid` parameter is ignored.

See also `PersistenceCapable.jdoNewInstance(StateManager sm)` and `PersistenceCapable.jdoNewInstance (StateManager sm, Object oid)`.

9.4 Registration of `PersistenceCapable` classes

```
public static void registerClass  
(Class pcClass, String[] fieldNames,  
Class[] fieldTypes,  
byte[] fieldFlags,  
Class persistenceCapableSuperclass,  
PersistenceCapable pcInstance);
```

This method registers a `PersistenceCapable` class so that the other methods can return the correct information. The registration must be done in a static initializer for the persistence-capable class.

9.4.1 Notification of `PersistenceCapable` class registrations

```
addRegisterClassListener(RegisterClassListener rcl);
```

This method registers a `RegisterClassListener` to be notified upon new `PersistenceCapable` Class registrations. A `RegisterClassEvent` instance is generated

for each class registered already plus classes registered in future, which is sent to each registered listener. The same event instance might be sent to multiple listeners.

```
removeRegisterClassListener(RegisterClassListener rcl);
```

This method removes a RegisterClassEvent from the list to be notified upon new PersistenceCapable Class registrations.

RegisterClassEvent

```
public class RegisterClassEvent extends java.util.EventObject {
```

An instance of this class is generated for each class that registers itself, and is sent to each registered listener.

```
public Class getRegisteredClass();
```

Returns the newly registered Class.

```
public String[] getFieldNames();
```

Returns the field names of the newly registered Class.

```
public Class[] getFieldTypes();
```

Returns the field types of the newly registered Class.

```
public byte[] getFieldFlags();
```

Returns the field flags of the newly registered Class.

```
public Class getPersistenceCapableSuperclass();
```

Returns the PersistenceCapable superclass of the newly registered Class.

```
} // class RegisterClassEvent
```

RegisterClassListener

```
interface RegisterClassListener extends java.util.EventListener {
```

This interface must be implemented by classes that register as listeners to be notified of registrations of PersistenceCapable classes.

```
void registerClass (RegisterClassEvent rce);
```

This method is called for each PersistenceCapable class that registers itself.

```
} // interface RegisterClassListener
```

9.5 Security administration

```
public static void registerAuthorizedStateManagerClass  
(Class smClass);
```

This method manages the list of classes authorized to execute replaceStateManager. During execution of this method, the security manager, if present, is called to validate that the caller is authorized for JDOPermission("setStateManager"). If successful, the parameter class is added to the list of authorized StateManager classes.

This method provides for a fast security check during makePersistent. An implementation of StateManager should register itself with the JDOImplHelper to take advantage of this fast check.

```
public static void checkAuthorizedStateManager(StateManager sm);
```

This method is called by enhanced persistence-capable class method `replaceStateManager`. If the parameter instance is of a class in the list of authorized `StateManager` classes, then this method returns silently. If not, then the security manager, if present, is called to validate that the caller is authorized for `JDOPermission("setStateManager")`. If successful, the method returns silently. If not, a `SecurityException` is thrown.

9.6 Application identity handling

```
public Object newObjectIdInstance(Class pcClass);
```

This method creates a new instance of the `Object Id` class for the `PersistenceCapable` class. If the class uses datastore identity, then `null` is returned. If the class is abstract, a `JDOFatalInternalException` is thrown.

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

This method creates a new instance of the `Object Id` class for the `PersistenceCapable` class, using the `String` constructor of the object id class. If the class uses datastore identity, then `null` is returned. If the class is abstract, a `JDOFatalInternalException` is thrown.

```
public void copyKeyFieldsToObjectId (Class pcClass, PersistenceCapable.ObjectIdFieldSupplier fs, Object oid);
```

This method copies key fields from the field manager to the `Object Id` instance `oid`. This is intended for use by the implementation to copy fields from a datastore-specific representation to the `Object Id`. If the class is abstract, a `JDOFatalInternalException` is thrown.

```
public void copyKeyFieldsFromObjectId (Class pcClass, PersistenceCapable.ObjectIdFieldConsumer fc, Object oid);
```

This method copies key fields to the field manager from the `Object Id` instance `oid`. This is intended for use by the implementation to copy fields to a datastore-specific representation from the `Object Id`. If the class is abstract, a `JDOFatalInternalException` is thrown.

10 InstanceCallbacks

Instance callbacks provide a mechanism for instances to take some action on specific JDO instance life cycle events. For example, classes that include non-persistent fields might use callbacks to correctly populate the values in these fields. Classes that affect the runtime environment might use callbacks to register and deregister themselves with other objects. This interface defines the methods executed by the `StateManager` for these life cycle events.

These methods will be called only on instances for which `javax.jdo.InstanceCallbacks.class.isInstance(pc)` returns `true`.

10.1 `jdoPostLoad`

```
public void jdoPostLoad();
```

This method is called after the default fetch group values have been loaded from the `StateManager` into the instance. Non-persistent fields whose value depends on values of default fetch group fields should be initialized in this method. This method is not modified by the enhancer. Only fields that are in the default fetch group should be accessed by this method, as other fields are not guaranteed to be initialized. This method might register the instance with other objects in the runtime environment.

The context in which this call is made does not allow access to other persistent JDO instances.

10.2 `jdoPreStore`

```
public void jdoPreStore();
```

This method is called before the values are stored from the instance to the datastore. This happens during `beforeCompletion` for persistent-new and persistent-dirty instances of persistence-capable classes that implement `InstanceCallbacks`. Datastore fields that might have been affected by modified non-persistent fields should be updated in this method. This method is modified by the enhancer so that changes to persistent fields will be reflected in the datastore.

The context in which this call is made allows access to the `PersistenceManager` and other persistent JDO instances.

This method is not called for deleted instances.

10.3 `jdoPreClear`

```
public void jdoPreClear();
```

This method is called before the implementation clears the values in the instance to their Java default values. This happens during an application call to `evict`, and in `afterCompletion` for `commit` with `RetainValues false` and `rollback` with `RestoreValues`

`false`. The method is called during any state transition to `hollow`. Non-persistent, non-transactional fields should be cleared in this method. Associations between this instance and others in the runtime environment should be cleared. This method is not modified by the enhancer, so access to fields is not mediated.

10.4 jdoPreDelete

```
public void jdoPreDelete();
```

This method is called during the execution of `deletePersistent` before the state transition to `persistent-deleted` or `persistent-new-deleted`. Access to field values within this call are valid. Access to field values after this call are disallowed. This method is modified by the enhancer so that fields referenced can be used in the business logic of the method.

To implement a containment aggregate, the user could implement this method to delete contained persistent instances.

11 PersistenceManagerFactory

This chapter details the `PersistenceManagerFactory`, which is responsible for creating `PersistenceManager` instances for application use.

```
package javax.jdo;

interface PersistenceManagerFactory {
```

11.1 Interface `PersistenceManagerFactory`

A JDO vendor must provide a class that implements `PersistenceManagerFactory` and is permitted to provide a `PersistenceManager` constructor[s].

A non-managed JDO application might choose to use a `PersistenceManager` constructor (JDO vendor specific) or use a `PersistenceManagerFactory` (provided by the JDO vendor). A portable JDO application must use the `PersistenceManagerFactory`.

In a managed environment, the JDO `PersistenceManager` instance is acquired by a two step process: the application uses JNDI lookup to retrieve an environment-named object, which is then cast to `javax.jdo.PersistenceManagerFactory`; and then calls one of the factory's `getPersistenceManager` methods.

In a non-managed environment, the JDO `PersistenceManager` instance is acquired by lookup as above; by constructing a `javax.jdo.PersistenceManager`; or by constructing a `javax.jdo.PersistenceManagerFactory`, configuring the factory, and then calling the factory's `getPersistenceManager` method. These constructors are not part of the JDO standard. However, the following is recommended to support portable applications.

Configuring the `PersistenceManagerFactory` follows the Java Beans pattern. Supported properties have a `get` method and a `set` method.

The following properties, if set in the `PersistenceManagerFactory`, are the default settings of all `PersistenceManager` instances created by the factory:

- `Optimistic`: the transaction mode that specifies concurrency control
- `RetainValues`: the transaction mode that specifies the treatment of persistent instances after commit
- `RestoreValues`: the transaction mode that specifies the treatment of persistent instances after rollback
- `IgnoreCache`: the query mode that specifies whether cached instances are considered when evaluating the filter expression
- `NontransactionalRead`: the `PersistenceManager` mode that allows instances to be read outside a transaction

- `NontransactionalWrite`: the `PersistenceManager` mode that allows instances to be written outside a transaction
- `Multithreaded`: the `PersistenceManager` mode that indicates that the application will invoke methods or access fields of managed instances from multiple threads.

The following properties are for convenience, if there is no connection pooling or other need for a connection factory:

- `ConnectionUserName`: the name of the user establishing the connection
- `ConnectionPassword`: the password for the user
- `ConnectionURL`: the URL for the data source
- `ConnectionDriverName`: the class name of the driver

For a portable application, if any other connection properties are required, then a connection factory must be configured.

The following properties are for use when a connection factory is used, and override the connection properties specified in `ConnectionURL`, `ConnectionUserName`, or `ConnectionPassword`.

- `ConnectionFactory`: the connection factory from which datastore connections are obtained
- `ConnectionFactoryName`: the name of the connection factory from which datastore connections are obtained. This name is looked up with JNDI to locate the connection factory.

If multiple connection properties are set, then they are evaluated in order:

- if `ConnectionFactory` is specified (not null), all other properties are ignored;
- else if `ConnectionFactoryName` is specified (not null), all other properties are ignored.

For the application server environment, connection factories always return connections that are enlisted in the thread's current transaction context. To use optimistic transactions in this environment requires a connection factory that returns connections that are not enlisted in the current transaction context. For this purpose, the following two properties are used:

- `ConnectionFactory2`: the connection factory from which nontransactional datastore connections are obtained
- `ConnectionFactory2Name`: the name of the connection factory from which nontransactional datastore connections are obtained. This name is looked up with JNDI to locate the connection factory.

Construction by Properties

An implementation must provide a method to construct a `PersistenceManagerFactory` by a `Properties` instance. This static method is called by the `JDOHelper` method `getPersistenceManagerFactory (Properties props)`.

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

The properties consist of: `"javax.jdo.PersistenceManagerFactoryClass"`, whose value is the name of the implementation class; any JDO vendor-specific properties;

and the following standard property names, which correspond to the properties as documented in this chapter:

- "javax.jdo.option.Optimistic"
- "javax.jdo.option.RetainValues"
- "javax.jdo.option.RestoreValues"
- "javax.jdo.option.IgnoreCache"
- "javax.jdo.option.NontransactionalRead"
- "javax.jdo.option.NontransactionalWrite"
- "javax.jdo.option.Multithreaded"
- "javax.jdo.option.ConnectionUserName"
- "javax.jdo.option.ConnectionPassword"
- "javax.jdo.option.ConnectionURL"
- "javax.jdo.option.ConnectionDriverName"
- "javax.jdo.option.ConnectionFactoryName"
- "javax.jdo.option.ConnectionFactory2Name"

The property "javax.jdo.PersistenceManagerFactoryClass" is the fully qualified class name of the `PersistenceManagerFactory`.

The `String` type properties are taken without change from the value of the corresponding keys. `Boolean` type properties treat the `String` value as representing `true` if the value of the `String` compares equal, ignoring case, to "true", and `false` if the value of the `String` is anything else.

Any property not recognized by the implementation must be silently ignored. Any standard property corresponding to an optional feature not supported by the implementation must throw `JDOUnsupportedOptionException`.

Default values for properties not specified in the `Properties` parameter are provided by the implementation. A portable application must specify all values for properties needed by the application.

11.2 ConnectionFactory

For implementations that layer on top of standard `Connector` implementations, the configuration will typically support all of the associated `ConnectionFactory` properties.

When used in a managed environment, the `ConnectionFactory` will be obtained from a `ManagedConnectionFactory`, which is then responsible for implementing the resource adapter interactions with the container.

The following properties of the `ConnectionFactory` should be used if the data source has a corresponding concept:

- `URL`: the URL for the data source
- `UserName`: the name of the user establishing the connection
- `Password`: the password for the user
- `DriverName`: the driver name for the connection

- **ServerName**: name of the server for the data source
- **PortNumber**: port number for establishing connection to the data source
- **MaxPool**: the maximum number of connections in the connection pool
- **MinPool**: the minimum number of connections in the connection pool
- **MsWait**: the number of milliseconds to wait for an available connection from the connection pool before throwing a `JDODataStoreException`
- **LogWriter**: the `PrintWriter` to which messages should be sent
- **LoginTimeout**: the number of seconds to wait for a new connection to be established to the data source

In addition to these properties, the `PersistenceManagerFactory` implementation class can support properties specific to the data source or to the `PersistenceManager`.

Aside from vendor-specific configuration APIs, there are three required methods for `PersistenceManagerFactory`.

11.3 PersistenceManager access

```
PersistenceManager getPersistenceManager();  
PersistenceManager getPersistenceManager(String userid, String  
password);
```

Returns a `PersistenceManager` instance with the configured properties. The instance might have come from a pool of instances. The default values for option settings are reset to the value specified in the `PersistenceManagerFactory` before returning the instance.

After the first use of `getPersistenceManager`, none of the `set` methods will succeed. The settings of operational parameters might be modified dynamically during runtime via a vendor-specific interface.

If the method with the `userid` and `password` is used to acquire the `PersistenceManager`, then all accesses to the connection factory during the life of the `PersistenceManager` will use the `userid` and `password` to get connections. If `PersistenceManager` instances are pooled, then only `PersistenceManager` instances with the same `userid` and `password` will be used to satisfy the request.

11.4 Close the PersistenceManagerFactory

During operation of JDO, resources might be acquired on behalf of a `PersistenceManagerFactory`, e.g. connection pools, persistence manager pools, compiled queries, cached metadata, etc. If a `PersistenceManagerFactory` is no longer needed, these resources should be returned to the system. The `close` method disables the `PersistenceManagerFactory` and allows cleanup of resources.

Premature close of a `PersistenceManagerFactory` has a significant impact on the operation of the system. Therefore, a security check is performed to check that the caller has the proper permission. The security check is for `JDOPermission("closePersistenceManagerFactory")`. If the security check fails, the `close` method throws `SecurityException`.

```
void close();
```

Close this `PersistenceManagerFactory`. Check for `JDOPermission("closePersistenceManagerFactory")` and if not authorized, throw `SecurityException`.

If the authorization check succeeds, check to see that all `PersistenceManager` instances obtained from this `PersistenceManagerFactory` have no active transactions. If any `PersistenceManager` instances have an active transaction, throw a `JDOUserException`, with one nested `JDOUserException` for each `PersistenceManager` with an active `Transaction`.

If there are no active transactions, then close all `PersistenceManager` instances obtained from this `PersistenceManagerFactory`, mark this `PersistenceManagerFactory` as closed, disallow `getPersistenceManager` methods, and allow all other `get` methods. If a `set` method or `getPersistenceManager` method is called after close, then `JDOUserException` is thrown.

11.5 Non-configurable Properties

The JDO vendor might store certain non-configurable properties and make those properties available to the application via a `Properties` instance. This method retrieves the `Properties` instance.

```
Properties getProperties();
```

The application is not prevented from modifying the instance.

Each key and value is a `String`. The keys defined for standard JDO implementations are:

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

Other properties are vendor-specific.

11.6 Optional Feature Support

```
Collection supportedOptions();
```

The JDO implementation might optionally support certain features, and will report the features that are supported. The supported query languages are included in the returned `Collection`.

This method returns a `Collection` of `String`, each `String` instance representing an optional feature of the implementation or a supported query language. The following are the values of the `String` for each optional feature in the JDO specification:

```
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
```

The standard JDO query must be returned as the `String`:

```
javax.jdo.query.JDOQL
```

Other query languages are represented by a `String` not defined in this specification.

11.7 Static Properties constructor

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

This method returns an instance of `PersistenceManagerFactory` based on the properties in the parameter. It is used by `JDOHelper` to construct an instance of `PersistenceManagerFactory` based on user-specified properties.

The following are standard key values for the `Properties`:

```
javax.jdo.PersistenceManagerFactoryClass  
javax.jdo.option.Optimistic  
javax.jdo.option.RetainValues  
javax.jdo.option.RestoreValues  
javax.jdo.option.IgnoreCache  
javax.jdo.option.NontransactionalRead  
javax.jdo.option.NontransactionalWrite  
javax.jdo.option.Multithreaded  
javax.jdo.option.ConnectionUserName  
javax.jdo.option.ConnectionPassword  
javax.jdo.option.ConnectionURL  
javax.jdo.option.ConnectionFactoryName  
javax.jdo.option.ConnectionFactory2Name
```

JDO implementations are permitted to define key values of their own. Any key values not recognized by the implementation must be ignored. Key values that are recognized but not supported by an implementation must result in a `JDOFatalUserException` thrown by the method.

The returned `PersistenceManagerFactory` is not configurable (the `setXXX` methods will throw an exception). JDO implementations might manage a map of instantiated `PersistenceManagerFactory` instances based on specified property key values, and return a previously instantiated `PersistenceManagerFactory` instance. In this case, the properties of the returned instance must exactly match the requested properties.

12 PersistenceManager

This chapter specifies the JDO `PersistenceManager` and its relationship to the application components, JDO instances, and J2EE Connector.

12.1 Overview

The JDO `PersistenceManager` is the primary interface for JDO-aware application components. It is the factory for the `Query` interface and contains methods for managing the life cycle of persistent instances.

The JDO `PersistenceManager` interface is architected to support a variety of environments and data sources, from small footprint embedded systems to large enterprise application servers. It might be a layer on top of a standard Connector implementation such as JDBC or JMS, or itself include connection management and distributed transaction support.

J2EE Connector support is optional. If it is not supported by a JDO implementation, then a constructor for the JDO `PersistenceManager` or `PersistenceManagerFactory` is required. The details of the construction of the `PersistenceManager` or `PersistenceManagerFactory` are not specified by JDO.

12.2 Goals

The architecture of the `PersistenceManager` has the following goals:

- No changes to application programs to change to a different vendor's `PersistenceManager` if the application is written to conform to the portability guidelines
- Application to non-managed and managed environments with no code changes

12.3 Architecture: JDO PersistenceManager

The JDO `PersistenceManager` instance is visible only to certain application components: those that explicitly manage the life cycle of JDO instances; and those that query for JDO instances. The JDO `PersistenceManager` is not required to be used by JDO instances.

There are three primary environments in which the JDO `PersistenceManager` is architected to work:

- non-managed (non-application server), minimum function, single transaction, single JDO `PersistenceManager` where compactness is the primary metric;
- non-managed but where extended features are desired, such as multiple `PersistenceManager` instances to support multiple data sources, XA coordinated transactions, or nested transactions; and

- managed, where the full range of capabilities of an application server is required.

Support for these three environments is accomplished by implementing transaction completion APIs on a companion JDO `Transaction` instance, which contains transaction policy options and local transaction support.

12.4 Threading

It is a requirement for all JDO implementations to be thread-safe. That is, the behavior of the implementation must be predictable in the presence of multiple application threads. Operations implemented by the `PersistenceManager` directly or indirectly via access or modification of persistent or transactional fields of persistence-capable classes must be treated as if they were serialized. The implementation is free to serialize internal data structures and thus order multi-threaded operations in any way it chooses. The only application-visible behavior is that operations might block indefinitely (but not infinitely) while other operations complete.

Since synchronizing the `PersistenceManager` is a relatively expensive operation, and not needed in many applications, the application must specify whether multiple threads might access the same `PersistenceManager` or instances managed by the `PersistenceManager` (persistent or transactional instances of `PersistenceCapable` classes; instances of `Transaction` or `Query`; query results, etc.).

If applications depend on serializing operations, then the applications must implement the appropriate synchronizing behavior, using instances visible to the application. This includes some instances of the JDO implementation (e.g. `PersistenceManager`, `Query`, etc.) and instances of persistence-capable classes.

The implementation must not use user-visible instances (instances of `PersistenceManagerFactory`, `PersistenceManager`, `Transaction`, `Query`, etc.) as synchronization objects, with one exception. The implementation must synchronize instances of `PersistenceCapable` during state transitions that replace the `StateManager`. This is to avoid race conditions where the application attempts to make the same instance persistent in multiple `PersistenceManagers`.

12.5 Class Loaders

JDO requires access to class instances in several situations where the class instance is not provided explicitly. In these cases, the only information available to the implementation is the name of the class.

To resolve class names to class instances, JDO implementations will use `Class.forName(String name, ClassLoader loader)` with up to three loaders. These loaders will be used in this order:

1. The loader that loaded the class or instance referred to in the API that caused this class to be loaded.

- In case of query, this is the loader of the candidate class.
- In case of navigation from a persistent instance, this is the loader of the class of the instance.
- In the case of `getExtent` with subclasses, this is the loader of the candidate class.
- In the case of `getObjectById`, this is the loader of the object id instance.
- Other cases do not have an explicit loader.

2. The loader returned in the current context by `Thread.getContextClassLoader()`.
3. The loader returned by `Thread.getContextClassLoader()` at the time of `PersistenceManagerFactory.getPersistenceManager()`. This loader is saved with the `PersistenceManager` and cleared when the `PersistenceManager` is closed.

12.6 Interface `PersistenceManager`

```
package javax.jdo;  
interface PersistenceManager {
```

A JDO `PersistenceManager` instance supports any number of JDO instances at a time. It is responsible for managing the identity of its associated JDO instances. A JDO instance is associated with either zero or one JDO `PersistenceManager`. It will be zero if and only if the JDO instance is in the transient state. As soon as the instance is made persistent or transactional, it will be associated with exactly one JDO `PersistenceManager`.

A JDO `PersistenceManager` instance supports one transaction at a time, and uses one connection to the underlying data source at a time. The JDO `PersistenceManager` instance might use multiple transactions serially, and might use multiple connections serially.

Therefore, to support multiple concurrent connection-oriented data sources in an application, multiple JDO `PersistenceManager` instances are required.

In this interface, JDO instances passed as parameters and returned as values must implement `PersistenceCapable`. However, the interface defines these formal parameters as `Object` because casting user classes to `PersistenceCapable` is awkward.

```
public interface javax.jdo.PersistenceManager {  
    boolean isClosed();  
    void close();
```

The `isClosed` method returns `false` upon construction of the `PersistenceManager` instance, or upon retrieval of a `PersistenceManager` from a pool. It returns `true` only after the `close` method completes successfully. After being closed, the `PersistenceManager` instance might be returned to the pool or garbage collected, at the choice of the JDO implementation. Before being used again to satisfy a `getPersistenceManager` request, the options will be reset to their default values as specified in the `PersistenceManagerFactory`.

In a non-managed environment, if the current transaction is active, `close` throws `JDOUserException`.

After `close` completes, all methods on the `PersistenceManager` instance except `isClosed` throw a `JDOFatalUserException`.

Null management

In the APIs that follow, `Object[]` and `Collection` are permitted parameter types. As these may contain nulls, the following rules apply.

Null arguments to APIs that take an `Object` parameter cause the API to have no effect. Null arguments to APIs that take `Object[]` or `Collection` will cause the API to throw `NullPointerException`. Non-null `Object[]` or `Collection` arguments that con-

tain null elements will have the documented behavior for non-null elements, and the null elements will be ignored.

12.6.1 Cache management

Normally, cache management is automatic and transparent. When instances are queried, navigated to, or modified, instantiation of instances and their fields and garbage collection of unreferenced instances occurs without any explicit control. When the transaction in which persistent instances are created, deleted, or modified completes, eviction is automatically done by the transaction completion mechanisms. Therefore, eviction is not normally required to be done explicitly. However, if the application chooses to become more involved in the management of the cache, several methods are available.

The non-parameter version of these methods applies the operation to each appropriate JDO instance in the cache. For `evictAll`, these are all persistent-clean instances; for `refreshAll`, all persistent-nontransactional instances.

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

Eviction is a hint to the `PersistenceManager` that the application no longer needs the parameter instances in the cache. Eviction allows the parameter instances to be subsequently garbage collected. Evicted instances will not have their values retained after transaction completion, regardless of the settings of the `retainValues` or `restoreValues` flags.

If `evictAll` with no parameters is called, then all persistent-clean instances are evicted (they transition to hollow). If users wish to automatically evict transactional instances at transaction commit time, then they should set `RetainValues` to false. Similarly, to automatically evict transactional instances at transaction rollback time, then they should set `RestoreValues` to false.

For each persistent-clean and persistent-nontransactional instance that the JDO `PersistenceManager` evicts, it:

- calls the `jdoPreClear` method on each instance, if the class of the instance implements `InstanceCallbacks`
- clears persistent fields on each instance (sets the value of the field to its Java default value);
- changes the state of instances to hollow.

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

The `refresh` method updates the values in the parameter instance[s] from the data in the datastore. The intended use is for optimistic transactions where the state of the JDO instance is not guaranteed to reflect the state in the datastore, and for datastore transactions to undo the changes to a specific set of instances instead of rolling back the entire transaction. This method can be used to minimize the occurrence of commit failures due to mismatch between the state of cached instances and the state of data in the datastore.

The `refreshAll` method with no parameters causes all transactional instances to be refreshed. If a transaction is not in progress, then this call has no effect.

Note that this method will cause loss of changes made to affected instances by the application due to refreshing the contents from the datastore.

The JDO `PersistenceManager`:

- loads persistent values from the datastore into the instance;
- calls the `jdoPostLoad` method on each persistent instance, if the class of the instance implements `InstanceCallbacks`; and
- changes the state of persistent-dirty instances to persistent-clean in a datastore transaction; or persistent-nontransactional in an optimistic transaction.

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

These methods request the `PersistenceManager` to load all persistent fields into the parameter instances. Subsequent to this call, the application might call `makeTransient` on the parameter instances, and the fields can no longer be touched by the `PersistenceManager`. The `PersistenceManager` might also retrieve related instances according to a pre-read policy (not specified by JDO).

If the `DFGOnly` parameter is `true`, then this is a hint to the implementation that only the fields in the default fetch group need to be retrieved. A compliant implementation is permitted to retrieve all fields regardless of the setting of this parameter. After the call with the `DFGOnly` parameter `true`, all default fetch group fields have been fetched, but other fields might be fetched lazily by the implementation.

The JDO `PersistenceManager`:

- loads persistent values from the datastore into the instance;
- for hollow instances, changes the state to persistent-clean in a datastore transaction; or persistent-nontransactional in an optimistic transaction, and if the class of the instance implements `InstanceCallbacks` calls `jdoPostLoad`.

12.6.2 Transaction factory interface

```
Transaction currentTransaction();
```

The `currentTransaction` method returns the `Transaction` instance associated with the `PersistenceManager`. The identical `Transaction` instance will be returned by all `currentTransaction` calls to the same `PersistenceManager` until `close`. Note that multiple transactions can be begun and completed (serially) with this same instance.

Even if the `Transaction` instance returned cannot be used for transaction completion (due to external transaction management), it still can be used to set flags.

12.6.3 Query factory interface

The query factory methods are detailed in the Query chapter .

```
void setIgnoreCache (boolean flag);
```

```
boolean getIgnoreCache ();
```

These methods get and set the value of the `IgnoreCache` option for all `Query` instances created by this `PersistenceManager` [see `Query` options]. The `IgnoreCache` option if set to `true`, is a hint to the query engine that the user expects queries to be optimized to return approximate results by ignoring changed values in the cache.

The `IgnoreCache` option also affects the iterator obtained from `Extent` instances obtained from this `PersistenceManager`.

The `IgnoreCache` option is preserved for query instances constructed from other query instances.

12.6.4 Extent Management

Extents are collections of datastore objects managed by the datastore, not by explicit user operations on collections. Extent capability is a boolean property of classes that are persistence capable. If an instance of a class that has a managed extent is made persistent via reachability, the instance is put into the extent implicitly.

```
Extent getExtent (Class PersistenceCapableClass, boolean sub-  
classes);
```

The `getExtent` method returns an `Extent` that contains all of the instances in the parameter class, and if the subclasses flag is `true`, all of the instances of the parameter class and its subclasses.

If the metadata does not indicate via the `requires-extent` attribute in the class element that an extent is managed for the parameter class, then `JDOUserException` is thrown. The extent might not include instances of those subclasses for which the metadata indicates that an extent is not managed for the subclass.

This method can be called whether or not a transaction is active, regardless of whether `NontransactionalRead` is supported. If `NontransactionalRead` is not supported, then the iterator method will throw a `JDOUnsupportedOptionException` if called outside a transaction.

It might be a common usage to iterate over the contents of the `Extent`, and the `Extent` should be implemented in such a way as to avoid out-of-memory conditions on iteration.

The primary use for the `Extent` returned as a result of this method is as a candidate collection parameter to a `Query` instance. For this usage, the elements in the `Extent` typically will not be instantiated in the Java VM; it is used only to identify the prospective datastore instances.

12.6.5 JDO Identity management

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

The `getObjectById` method attempts to find an instance in the cache with the specified JDO identity. The `oid` parameter object might have been returned by an earlier call to `getObjectId` or `getTransactionalObjectId`, or might have been constructed by the application.

If the `PersistenceManager` is unable to resolve the `oid` parameter to an `ObjectId` instance, then it throws a `JDOUserException`. This might occur if the implementation does not support application identity, and the parameter is an instance of an object identity class.

- If the `validate` flag is `false`:

- If there is already an instance in the cache with the same JDO identity as the oid parameter, then this method returns it. There is no change made to the state of the returned instance.
- If there is not an instance already in the cache with the same JDO identity as the oid parameter, then this method creates an instance with the specified JDO identity and returns it. If there is no transaction in progress, the returned instance will be hollow or persistent-nontransactional, at the choice of the implementation.
- If there is a transaction in progress, the returned instance will be hollow, persistent-nontransactional, or persistent-clean, at the choice of the implementation.
- It is an implementation decision whether to access the datastore, if required to determine the exact class. This will be the case of inheritance, where multiple `PersistenceCapable` classes share the same `Object Id` class.
- If the instance does not exist in the datastore, then this method might not fail. It is an implementation choice if the method fails immediately with a `JDOObjectNotFoundException`. But a subsequent access of the fields of the instance will throw a `JDOObjectNotFoundException` if the instance does not exist at that time. Further, if a relationship is established to this instance, and the instance does not exist when the instance is flushed to the datastore, then the transaction in which the association was made will fail.
- If the `validate` flag is true:
 - If there is already a transactional instance in the cache with the same jdo identity as the oid parameter, then this method returns it. There is no change made to the state of the returned instance.
 - If there is an instance already in the cache with the same jdo identity as the oid parameter, the instance is not transactional, and the instance does not exist in the datastore, then a `JDOObjectNotFoundException` is thrown.
 - If there is not an instance already in the cache with the same jdo identity as the oid parameter, then this method creates an instance with the specified jdo identity, verifies that it exists in the datastore, and returns it. If the instance does not exist in the datastore, then a `JDOObjectNotFoundException` is thrown.
 - If there is no transaction in progress, the returned instance will be hollow or persistent-nontransactional, at the choice of the implementation.
 - If there is a datastore transaction in progress, the returned instance will be persistent-clean.
 - If there is an optimistic transaction in progress, the returned instance will be persistent-nontransactional.

`Object getObjectId (Object pc);`

The `getObjectId` method returns an `ObjectId` instance that represents the object identity of the specified JDO instance. The identity is guaranteed to be unique only in the context of the JDO `PersistenceManager` that created the identity, and only for two types of JDO Identity: those that are managed by the application, and those that are managed by the datastore.

If the object identity is being changed in the transaction, by the application modifying one or more of the application key fields, then this method returns the identity as of the beginning of the transaction. The value returned by `getObjectId` will be different following `afterCompletion` processing for successful transactions.

Within a transaction, the `ObjectId` returned will compare equal to the `ObjectId` returned by only one among all JDO instances associated with the `PersistenceManager` regardless of the type of `ObjectId`.

The `ObjectId` does not necessarily contain any internal state of the instance, nor is it necessarily an instance of the class used to manage identity internally. Therefore, if the application makes a change to the `ObjectId` instance returned by this method, there is no effect on the instance from which the `ObjectId` was obtained.

The `getObjectById` method can be used between instances of `PersistenceManager` of different JDO vendors only for instances of persistence capable classes using application-managed (primary key) JDO identity. If it is used for instances of classes using datastore identity, the method might succeed, but there are no guarantees that the parameter and return instances are related in any way.

If the parameter `pc` is not persistent, or is null, then null is returned.

```
Object getTransactionalObjectId (Object pc);
```

If the object identity is being changed in the transaction, by the application modifying one or more of the application key fields, then this method returns the current identity in the transaction. If there is no transaction in progress, or if none of the key fields is being modified, then this method has the same behavior as `getObjectId`.

To get an instance in a `PersistenceManager` with the same identity as an instance from a different `PersistenceManager`, use the following: `aPersistenceManager.getObjectById(JDOHelper.getObjectId(pc), validate)`. The `validate` parameter has a value of `true` or `false` depending on your application requirements.

12.6.6 JDO Instance life cycle management

The following methods take either a single instance or multiple instances as parameters.

If a collection or array of instances is passed to any of the methods in this section, and one or more of the instances fail to complete the required operation, then all instances will be attempted, and a `JDOUserException` will be thrown which contains a nested exception array, each exception of which contains one of the failing instances. The succeeding instances will transition to the specified life cycle state, and the failing instances will remain in their current state.

Make instances persistent

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

These methods make a transient instance persistent directly. They must be called in the context of an active transaction, or a `JDOUserException` is thrown. They will assign an object identity to the instance and transition it to `persistent-new`. Any transient instances reachable from this instance via persistent fields of this instance will become provisionally persistent, transitively. That is, they behave as `persistent-new` instances (return `true` to `isPersistent`, `isNew`, and `isDirty`). But at commit time, the reachability algorithm is run again, and instances made provisionally persistent that are not currently reachable from persistent instances will revert to transient.

These methods have no effect on parameter persistent instances already managed by this `PersistenceManager`. They will throw a `JDOUserException` if the parameter instance is managed by a different `PersistenceManager`.

If an instance is of a class whose identity type (application, datastore, or none) is not supported by the JDO implementation, then a `JDOUserException` will be thrown for that instance.

Delete persistent instances

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

These methods delete persistent instances from the datastore. They must be called in the context of an active transaction, or a `JDOUserException` is thrown. The representation in the datastore will be deleted when this instance is flushed to the datastore (via `commit` or `evict`).

Note that this behavior is not exactly the inverse of `makePersistent`, due to the transitive nature of `makePersistent`. The implementation might delete dependent datastore objects depending on implementation-specific policy options that are not covered by the JDO specification.

These methods have no effect on parameter instances already deleted in the transaction or on embedded instances. Embedded instances are deleted when their owning instance is deleted.

If deleting an instance would violate datastore integrity constraints, it is implementation-defined whether an exception is thrown at commit time, or the delete operation is simply ignored. Portable applications should use this method to delete instances from the datastore, and not depend on any reachability algorithm to automatically delete instances.

These methods will throw a `JDOUserException` if the parameter instance is managed by a different `PersistenceManager`. These methods will throw a `JDOUserException` if the parameter instance is transient.

Make instances transient

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

These methods make persistent instances transient, so they are no longer associated with the `PersistenceManager` instance. They do not affect the persistent state in the datastore. They can be used as part of a sequence of operations to move a persistent instance to another `PersistenceManager`. The instance transitions to transient, and it loses its JDO identity. If the instance has state (persistent-nontransactional or persistent-clean) the state in the cache is preserved unchanged. If the instance is dirty, a `JDOUserException` is thrown.

The effect of this method is immediate and not subject to rollback. Field values in the instances are not modified. To avoid having the instances become persistent by reachability at commit, the application should update all persistent instances containing references to the parameter instances to avoid referring to them, or make the referring instances transient.

These methods will be ignored if the instance is transient.

Make instances transactional

```
void makeTransactional (Object pc);
```

```
void makeTransactionalAll (Object[] pcs);  
void makeTransactionalAll (Collection pcs);
```

These methods make transient instances transactional and cause a state transition to transient-clean. After the method completes, the instance observes transaction boundaries. If the transaction in which this instance is made transactional commits, then the transient instance retains its values. If the transaction is rolled back, then the transient instance takes its values as of the call to `makeTransactional` if the call was made within the current transaction; or the beginning of the transaction, if the call was made prior to the beginning of the current transaction.

If the implementation does not support `TransientTransactional`, and the parameter instance is transient, then `JDOUnsupportedOperationException` is thrown.

These methods are also used to mark a nontransactional persistent instance as being part of the read-consistency set of the transaction. In this case, the call must be made in the context of an active transaction, or a `JDOUserException` is thrown.

The effect of these methods is immediate and not subject to rollback.

Make instances nontransactional

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

These methods make transient-clean instances nontransactional and cause a state transition to transient. After the method completes, the instance does not observe transaction boundaries.

These methods make persistent-clean instances nontransactional and cause a state transition to persistent-nontransactional.

If this method is called with a dirty parameter instance, a `JDOUserException` is thrown.

The effect of these methods is immediate and not subject to rollback.

12.7 Transaction completion

Transaction completion management is delegated to the associated `Transaction` instance .

12.8 Multithreaded Synchronization

The application might require the `PersistenceManager` to synchronize internally to avoid corruption of data structures due to multiple application threads. This synchronization is not required when the flag `Multithreaded` is set to `false`.

```
void setMultithreaded (boolean flag);  
boolean getMultithreaded();
```

NOTE: When the `Multithreaded` flag is set to `true`, there is a synchronization issue with `jdoFlags` values `READ_OK` and `READ_WRITE_OK`. Due to out-of-order memory writes, there is a chance that a value for a field in the default fetch group might be incorrect (stale) when accessed by a thread that has not synchronized with the thread that set the `jdoFlags` value. Therefore, it is recommended that a JDO implementation not use `READ_OK` or `READ_WRITE_OK` for `jdoFlags` if `Multithreaded` is set to `true`.

The application may choose to perform its own synchronization, and indicate this to the implementation by setting the `Multithreaded` flag to `false`. In this case, the JDO implementation is not required to implement any additional synchronizations, although it is permitted to do so.

12.9 User associated object

The application might manage `PersistenceManager` instances by using an associated object for bookkeeping purposes. These methods allow the user to manage the associated object.

```
void setUserObject (Object o);  
Object getUserObject ();
```

The parameter is not inspected or used in any way by the JDO implementation.

12.10 PersistenceManagerFactory

The application might need to get the `PersistenceManagerFactory` that created this `PersistenceManager`. If the `PersistenceManager` was created using a constructor, then this call returns `null`.

```
PersistenceManagerFactory getPersistenceManagerFactory();
```

12.11 ObjectId class management

In order for the application to construct instances of the `ObjectId` class, there is a method that returns the `ObjectId` class given the persistence capable class.

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

This method returns the class of the object id for the given class. This method returns the class specified by the application for persistence capable classes that use application (primary key) JDO identity. It returns the implementation-defined class for persistence-capable classes that use datastore identity. If the parameter class is not persistence-capable, or the parameter is `null`, `null` is returned. If the object-id class defined in the metadata for the parameter class is abstract then `null` is returned.

If the implementation does not support application identity, and the class is defined in the jdo metadata to use application identity, then `null` is returned.

```
Object newObjectIdInstance (Class pcClass, String str);
```

This method returns an object id instance corresponding to the `Class` and `String` arguments. The `String` argument might have been the result of executing `toString` on an object id instance.

This method is portable for datastore identity and application identity.

13 Transactions and Connections

This chapter describes the interactions among JDO instances, JDO Persistence Managers, datastore transactions, and datastore connections.

13.1 Overview

Operations on persistent JDO instances at the user's choice might be performed in the context of a transaction. That is, the view of data in the datastore is transactionally consistent, according to the standard definition of ACID transactions:

- atomic -- within a transaction, changes to values in JDO instances are all executed or none is executed
- consistent -- changes to values in JDO instances are consistent with changes to other values in the same JDO instance
- isolated -- changes to values in JDO instances are isolated from changes to the same JDO instances in different transactions
- durable -- changes to values in JDO instances survive the end of the VM in which the changes were made

13.2 Goals

The JDO transaction and connection contracts have the following goals.

- JDO implementations might span a range of small, embedded systems to large, enterprise systems
- Transaction management might be entirely hidden from class developers and application components, or might be explicitly exposed to class and application component developers.

13.3 Architecture: PersistenceManager, Transactions, and Connections

An instance of an object supporting the `PersistenceManager` interface represents a single user's view of persistent data, including cached persistent instances across multiple serial datastore transactions.

There is a one-to-one relationship between the `PersistenceManager` and the `Transaction`. The `Transaction` interface is isolated because of separation of concerns. The methods could have been added to the `PersistenceManager` interface.

The `javax.jdo.Transaction` interface provides for management of transaction options and, in the non-managed environment, for transaction completion. It is similar in functionality to `javax.transaction.UserTransaction`. That is, it contains `begin`, `commit`, and `rollback` methods used to delimit transactions.

Connection Management Scenarios

- **single connection:** In the simplest case, the `PersistenceManager` directly connects to the datastore and manages transactional data. In this case, there is no reason to expose any `Connection` properties other than those needed to identify the user and the data source. During transaction processing, the `Connection` will be used to satisfy data read, write, and transaction completion requests from the `PersistenceManager`.
- **connection pooling:** In a slightly more complex situation, the `PersistenceManagerFactory` creates multiple `PersistenceManager` instances which use connection pooling to reduce resource consumption. The `PersistenceManagers` are used in single datastore transactions. In this case, a pooling connection manager is a separate component used by the `PersistenceManager` instances to effect the pooling of connections. The `PersistenceManagerFactory` will include a reference to the connection pooling component, either as a JNDI name or as an object reference. The connection pooling component is separately configured, and the `PersistenceManagerFactory` simply needs to be configured to use it.
- **distributed transactions:** An even more complex case is where the `PersistenceManager` instances need to use connections that are involved in distributed transactions. This case requires coordination with a `Transaction Manager`, and exposure of the `XAResource` from the datastore `Connection`. JDO does not specify how the application coordinates transactions among the `PersistenceManager` and the `Transaction Manager`.
- **managed connections:** The last case to consider is the managed environment, where the `PersistenceManagerFactory` uses a datastore `Connection` whose transaction completion is managed by the application server. This case requires the datastore `Connection` to implement the J2EE Connector Architecture and the `PersistenceManager` to use the architected interfaces to obtain a reference to a `Connection`.

The interface between the JDO implementation and the `Connection` component is not specified by JDO. In the non-managed environment, transaction completion is handled by the `Connection` managed internally by the `Transaction`. In the managed environment, transaction completion is handled by the `XAResource` associated with the `Connection`. In both cases, the `PersistenceManager` implementation is responsible for setting up the appropriate interface to the `Connection` infrastructure.

Native Connection Management

If the JDO implementation supplies its own resource adapter implementation, this is termed native connection management. For use in a managed environment, the association between `Transaction` and `Connection` must be established using the J2EE Connector Architecture [see Appendix A reference 4]. This is done by the JDO implementation implementing the `javax.resource.ManagedConnectionFactory` interface.

When used in a non-managed environment, with non-distributed transaction management (local transactions) the application can use the `PersistenceManagerFactory`. But if distributed transaction management is required, the application needs to supply an implementation of `javax.resource.ManagedConnectionFactory` interface. This interface provides the infrastructure to enlist the `XAResource` with the `Transaction Manager` used in the application.

Non-native Connection Management

If the JDO implementation uses a third party Connection interface, then it can be used in a managed environment only if the third party Connection supports the J2EE Connector Architecture. In this case, the `PersistenceManagerFactory` property `ConnectionFactory` is used to allow the application server to manage connections.

In the non-managed case, non-distributed transaction management can use the `PersistenceManagerFactory`, as above. But if distributed transaction management is required, the application needs to supply an implementation of `javax.resource.ConnectionManager` interface to be used with the application's implementation of the Connection management.

Optimistic Transactions

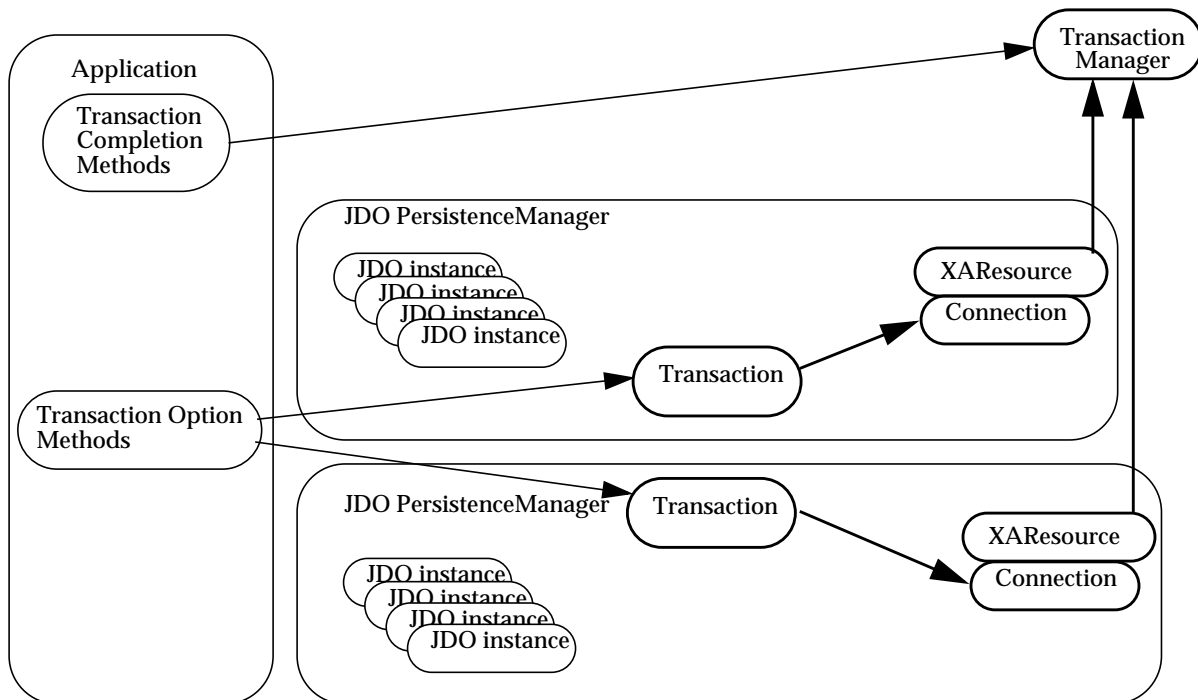
There are two types of transaction management strategies supported by JDO: “datastore transaction management”; and “optimistic transaction management”.

With datastore transaction management, all operations performed by the application on persistent data are done using a datastore transaction. This means that between the first data access until the commit, there is an active datastore transaction.

With optimistic transaction management, operations performed by the application on persistent data outside a transaction or before commit are done using a short local datastore transaction. During flush, a datastore transaction is used for the update operations, verifying that the proposed changes do not conflict with a parallel update by a different transaction.

Optimistic transaction management is specified by the `Optimistic` setting on `Transaction`.

Figure 15.0 Transactions and Connections



13.4 Interface Transaction

```
package javax.jdo.Transaction;  
interface Transaction {
```

13.4.1 PersistenceManager

```
PersistenceManager getPersistenceManager ();
```

This method returns the `PersistenceManager` associated with this `Transaction` instance.

```
boolean isActive ();
```

This method tells whether there is an active transaction. The transaction might be either a local transaction or a distributed transaction. If the transaction is local, then the `begin` method was executed and neither `commit` nor `rollback` has been executed. If the transaction is managed by `XAResource` with a `TransactionManager`, then this method indicates whether there is a distributed transaction active.

This method returns `true` after the transaction has been started, until the `afterCompletion` synchronization method is called.

13.4.2 Transaction options

Transaction options are valid for both managed and non-managed environments. Flags are durable until changed explicitly by `set` methods. They are not changed by transaction demarcation methods.

If any of the `set` methods is called during `commit` or `rollback` processing (within the `beforeCompletion` and `afterCompletion` synchronization methods), a `JDOUserException` is thrown.

If an implementation does not support the option, then an attempt to set the flag to an unsupported value will throw `JDOUnsupportedOptionException`.

Nontransactional access to persistent values

```
boolean getNontransactionalRead ();
```

```
void setNontransactionalRead (boolean flag);
```

These methods access the flag that allows persistent instances to be read outside a transaction. If this flag is set to `true`, then queries and field read access (including navigation) are allowed without an active transaction. If this flag is set to `false`, then queries and field read access (including navigation) outside an active transaction throw a `JDOUserException`.

```
boolean getNontransactionalWrite ();
```

```
void setNontransactionalWrite (boolean flag);
```

These methods access the flag that allows non-transactional instances to be written in the cache. If this flag is set to `true`, then updates to non-transactional instances are allowed without an active transaction. If this flag is set to `false`, then updates to non-transactional instances outside an active transaction throw a `JDOUserException`.

Optimistic concurrency control

If this flag is set to `true`, then optimistic concurrency is used for managing transactions.

```
boolean getOptimistic ();
```

The optimistic setting currently active is returned.

```
void setOptimistic (boolean flag);
```

The optimistic setting passed replaces the optimistic setting currently active.

This method can be used only when there is not an active transaction. If it is used while there is an active transaction, a `JDOUserException` is thrown.

Retain values at transaction commit

If this flag is set to `true`, then eviction of transactional persistent instances does not take place at transaction commit. If this flag is set to `false`, then eviction of transactional persistent instances takes place at transaction commit.

```
boolean getRetainValues ();
```

The `retainValues` setting currently active is returned.

```
void setRetainValues (boolean flag);
```

The `retainValues` setting passed replaces the `retainValues` setting currently active.

Restore values at transaction rollback

If this flag is set to `true`, then restoration of transactional persistent instances takes place at transaction rollback. If this flag is set to `false`, then eviction of transactional persistent instances takes place at transaction rollback.

```
boolean getRestoreValues ();
```

The `restoreValues` setting currently active is returned.

```
void setRestoreValues (boolean flag);
```

The `restoreValues` setting passed replaces the `restoreValues` setting currently active. This method can be used only when there is not an active transaction. If it is used while there is an active transaction, a `JDOUserException` is thrown.

13.4.3 Synchronization

The `Transaction` instance participates in synchronization in two ways: as a supplier of synchronization callbacks, and as a consumer of callbacks. As a supplier of callbacks, a user can register with the `Transaction` instance to be notified at transaction completion. As a consumer of callbacks, the `Transaction` implementation will use the proprietary interfaces of the managed environment to be notified of externally-initiated transaction completion events. In a managed environment, this notification is used to cause flushing of changes to the datastore as part of transaction completion.

For this latter purpose, the JDO implementation class might implement `javax.transaction.Synchronization` or might use a delegate to be notified.

Synchronization is supported for both managed and non-managed environments. A `Synchronization` instance registered with the `Transaction` remains registered until changed explicitly by another `setSynchronization`.

Only one `Synchronization` instance can be registered with the `Transaction`. If the application requires more than one instance to receive synchronization callbacks, then the application instance is responsible for managing them, and forwarding callbacks to them.

```
void setSynchronization (javax.transaction.Synchronization  
sync);
```

The `Synchronization` instance is registered with the `Transaction` for transaction completion notifications. Any `Synchronization` instance already registered will be re-

placed. If the parameter is null, then no instance will be notified. If this method is called during commit processing (within the user's `beforeCompletion` or `afterCompletion` method), a `JDOUserException` is thrown.

The `beforeCompletion` method will be called during the behavior specified for the transaction completion method `commit`. The `beforeCompletion` method will not be called before `rollback`.

The `afterCompletion` method will be called during the transaction completion methods. The parameter for the `afterCompletion(int status)` method will be either `javax.transaction.Status.STATUS_COMMITTED` or `javax.transaction.Status.STATUS_ROLLEDBACK`.

These two methods allow the application control over the environment in which the transaction completion executes (for example, validate the state of the cache before completion) and to control the cache disposition once the transaction completes (for example, to change persistent instances to persistent-nontransactional state).

```
javax.transaction.Synchronization getSynchronization ();
```

This method returns the `Synchronization` currently registered.

13.4.4 Transaction demarcation

If multiple parallel transactions are required, then multiple `PersistenceManager` instances must be used. If distributed transactions are required, then the Connector Architecture is used to coordinate transactions among the JDO `PersistenceManagers`.

Non-managed environment

In a non-managed environment, with a single JDO `PersistenceManager` per application, there is a `Transaction` instance representing a local transaction associated with the `PersistenceManager` instance.

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

The `commit` and `rollback` methods can be used only in a non-managed environment, or in a managed environment with Bean Managed Transactions. If one of these methods is executed in a managed environment with Container Managed Transactions, a `JDOUserException` is thrown.

The `commit` method performs the following operations:

- calls the `beforeCompletion` method of the `Synchronization` instance registered with the `Transaction`;
- flushes dirty persistent instances;
- notifies the underlying datastore to commit the transaction;
- transitions persistent instances according to the life cycle specification;
- calls the `afterCompletion` method of the `Synchronization` instance registered with the `Transaction` with the results of the datastore commit operation.

The `rollback` method performs the following operations:

- transitions persistent instances according to the life cycle specification;

- rolls back changes made in this transaction from the datastore;
- calls the `afterCompletion` method of the `Synchronization` instance registered with the `Transaction`.

Managed environment

In a managed environment, there is either a user transaction or a local transaction associated with the `PersistenceManager` instance when executing method calls on JDO instances or on the `PersistenceManager`. Which of the two types of transactions is active is a policy issue for the managed environment.

If datastore transaction management is being used with the `PersistenceManager` instance, and a `Connection` to the datastore is required during execution of the `PersistenceManager` or JDO instance method, then the `PersistenceManager` will dynamically acquire a `Connection`. The call to acquire the `Connection` will be made with the calling thread in the appropriate transactional context, and the `Connection` acquired will be in the proper datastore transaction.

If optimistic transaction management is being used with the `PersistenceManager` instance, and a `Connection` to the datastore is required during execution of an instance method or a non-completion `PersistenceManager` method, then the `PersistenceManager` will use a local transaction `Connection`.

13.5 Optimistic transaction management

Optimistic transactions are an optional feature of a JDO implementation. They are useful when there are long-running transactions that rarely affect the same instances, and therefore the datastore will exhibit better performance by deferring datastore exclusion on modified instances until commit.

In the following discussion, “transactional datastore context” refers to the transaction context of the underlying datastore, while “transaction”, “datastore transaction”, and “optimistic transaction” refer to the JDO transaction concepts.

With datastore transactions, persistent instances accessed within the scope of an active transaction are guaranteed to be associated with the transactional datastore context. With optimistic transactions, persistent instances accessed within the scope of an active transaction are not associated with the transactional datastore context; the only time any instances are associated with the transactional datastore context is during commit.

With optimistic transactions, instances queried or read from the datastore will not be transactional unless they are modified, deleted, or marked by the application as transactional. At commit time, the JDO implementation:

- establishes a transactional datastore context in which verification, insert, delete, and updates will take place.
- calls the `beforeCompletion` method of the `Synchronization` instance registered with the `Transaction`;
- verifies unmodified instances that have been made transactional, to ensure that the state in the datastore is the same as the instance used in the transaction [this is done using a JDO implementation-specific algorithm];

- verifies modified and deleted instances during flushing to the datastore, to ensure that the state in the datastore is the same as the before image of the instance that was modified or deleted by the transaction [this is done using a JDO implementation-specific algorithm]
- If any instance fails the verification, a `JDOOptimisticVerificationException` is thrown which contains an array of `JDOOptimisticVerificationException`, one for each instance that failed the verification. The optimistic transaction is failed, and the transaction is rolled back. The definition of “changed instance” is a JDO implementation choice, but it is required that a field that has been changed to different values in different transactions results in one of the transactions failing.
- if verification succeeds, notifies the underlying datastore to commit the transaction;
- transitions persistent instances according to the life cycle specification, based on whether the transaction succeeds and the setting of the `RetainValues` and `RestoreValues` flags;
- calls the `afterCompletion` method of the `Synchronization` instance registered with the `Transaction` with the results of the commit operation.

Details of the state transitions of persistent instances in optimistic transactions may be found in section 5.8.

14 Query

This chapter specifies the query contract between an application component and the `JDO PersistenceManager`.

The query facility consists of two parts: the query API, and the query language. The query language described in this chapter is “JDOQL”.

14.1 Overview

An application component requires access to JDO instances so it can invoke specific behavior on those instances. From a JDO instance, it might navigate to other associated instances, thereby operating on an application-specific closure of instances.

However, getting to the first JDO instance is a bootstrap issue. There are three ways to get an instance from JDO. First, if the users have or can construct a valid `ObjectId`, then they can get an instance via the persistence manager’s `getObjectById` method. Second, users can iterate a class extent by calling `getExtent`. Third, the `JDO Query` interface provides the ability to acquire access to JDO instances from a particular JDO persistence manager based on search criteria specified by the application.

The persistent manager instance is a factory for query instances, and queries are executed in the context of the persistent manager instance.

The actual query execution might be performed by the `JDO PersistenceManager` or might be delegated by the `JDO PersistenceManager` to its datastore. The actual query executed thus might be implemented in a very different language from Java, and might be optimized to take advantage of particular query language implementations.

For this reason, methods in the query filter have semantics possibly different from those in the Java VM.

14.2 Goals

The `JDO Query` interface has the following goals:

- Query language neutrality. The underlying query language might be a relational query language such as SQL; an object database query language such as OQL; or a specialized API to a hierarchical database or mainframe EIS system.
- Optimization to specific query language. The `Query` interface must be capable of optimizations; therefore, the interface must have enough user-specified information to allow for the JDO implementation to exploit data source specific query features.
- Accommodation of multi-tier architectures. Queries might be executed entirely in memory, or might be delegated to a back end query engine. The `JDO Query` interface must provide for both types of query execution strategies.

- Large result set support. Queries might return massive numbers of JDO instances that match the query. The JDO Query architecture must provide for processing the results within the resource constraints of the execution environment.
- Compiled query support. Parsing queries may be resource-intensive, and in many applications can be done during application development or deployment, prior to execution time. The query interface allows for compiling queries and binding run-time parameters to the bound queries for execution.

14.3 Architecture: Query

The JDO `PersistenceManager` instance is a factory for JDO Query instances, which implement the JDO Query interface. Multiple JDO Query instances might be active simultaneously in the same JDO `PersistenceManager` instance. Multiple queries might be executed simultaneously by different threads, but the implementation might choose to execute them serially. In either case, the execution must be thread safe.

There are three required elements in any query:

- the class of the candidate instances. The class is used to scope the names in the query filter. All of the candidate instances are of this class or a subclass of this class.
- the collection of candidate JDO instances. The collection of candidate instances is either a `java.util.Collection`, or an `Extent` of instances in the datastore. Instances that are not of the required class or subclass will be silently ignored. The `Collection` might be a previous query result, allowing for subqueries.
- the query filter. The query filter is a Java `boolean` expression that tells whether instances in the candidate collection are to be returned in the result. If not specified, the filter defaults to `true`.

Other elements in queries include:

- parameter declarations. The parameter declaration is a `String` containing one or more query parameter declarations separated with commas. It follows the syntax for formal parameters in the Java language. Each parameter named in the parameter declaration must be bound to a value when the query is executed.
- parameter values to bind to parameters. Values are specified as Java Objects, and might include simple wrapper types or more complex object types. The values are passed to the execute methods and are not preserved after a query executes.
- variable declarations: Variables might be used in the filter, and these variables must be declared with their type. The variable declaration is a `String` containing one or more variable declarations. Each declaration consists of a type and a variable name, with declarations separated by a semicolon if there are two or more declarations. It is similar to the syntax for local variables in the Java language.
- import statements: Parameters and variables might come from a different class from the candidate class, and the names might need to be declared in an import statement to eliminate ambiguity. Import statements are specified as a `String` with semicolon-separated statements. The syntax is the same as in the Java language import statement.
- ordering specification. The ordering specification includes a list of expressions with the ascending/descending indicator. The expression's type must be one of:

- primitive types except `boolean`;
- wrapper types except `Boolean`;
- `BigDecimal`;
- `BigInteger`;
- `String`;
- `Date`.

The class implementing the `Query` interface must be serializable. The serialized fields include the candidate class, the filter, parameter declarations, variable declarations, imports, and ordering specification. If a serialized instance is restored, it loses its association with its former `PersistenceManager`.

14.4 Namespaces in queries

The query namespace is modeled after methods in Java:

- `setClass` corresponds to the class definition
- `declareParameters` corresponds to formal parameters of a method
- `declareVariables` corresponds to local variables of a method
- `setFilter` and `setOrdering` correspond to the method body

There are two namespaces in queries. Type names have their own namespace that is separate from the namespace for fields, variables and parameters.

The method `setClass` introduces the name of the candidate class in the type namespace. The method `declareImports` introduces the names of the imported class or interface types in the type namespace. When used (e.g. in a parameter declaration, cast expression, etc.) a type name must be the name of the candidate class, the name of a class or interface imported by the parameter to `declareImports`, denote a class or interface from the same package as the candidate class, or must be declared by exactly one type-import-on-demand declaration (`"import <package>.*;"`). It is valid to specify the same import multiple times.

The names of the public types declared in the package `java.lang` are automatically imported as if the declaration `"import java.lang.*;"` appeared in `declareImports`. It is a JDOQL-compile time error (reported during `compile()` or `execute(...)` methods) if a used type name is declared by more than one type-import-on-demand declaration.

The method `setClass` also introduces the names of the candidate class fields.

The method `declareParameters` introduces the names of the parameters. A name introduced by `declareParameters` hides the name of a candidate class field if equal. Parameter names must be unique.

The method `declareVariables` introduces the names of the variables. A name introduced by `declareVariables` hides the name of a candidate class field if equal. Variable names must be unique and must not conflict with parameter names.

A hidden field may be accessed using the `this` qualifier: `this.fieldName`.

14.5 Query Factory in `PersistenceManager` interface

The `PersistenceManager` interface contains `Query` factory methods.

```
Query newQuery();
```

Construct an empty query instance.

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

Construct a query instance from another query. The parameter might be a serialized/restored `Query` instance from the same JDO vendor but a different execution environment, or the parameter might be currently bound to a `PersistenceManager` from the same JDO vendor. Any of the elements `Class`, `Filter`, `IgnoreCache` flag, `Import` declarations, `Variable` declarations, `Parameter` declarations, and `Ordering` from the parameter `Query` are copied to the new `Query` instance, but a candidate `Collection` or `Extent` element is discarded.

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

Construct a query instance using the specified language and the specified query. The query instance will be of a class defined by the query language. The language parameter for the JDO Query language as herein documented is `"javax.jdo.query.JDOQL"`. Other languages' parameter is not specified.

```
Query newQuery (Class cls);
```

Construct a query instance with the candidate class specified.

```
Query newQuery (Extent cln);
```

Construct a query instance with the candidate `Extent` specified; the candidate class is taken from the `Extent`.

```
Query newQuery (Class cls, Collection cln);
```

Construct a query instance with the candidate class and candidate `Collection` specified.

```
Query newQuery (Class cls, String filter);
```

Construct a query instance with the candidate class and filter specified.

```
Query newQuery (Class cls, Collection cln, String filter);
```

Construct a query instance with the candidate class, the candidate `Collection`, and filter specified.

```
Query newQuery (Extent cln, String filter);
```

Construct a query instance with the candidate `Extent` and filter specified; the candidate class is taken from the `Extent`.

14.6 Query Interface

```
package javax.jdo;
```

```
interface Query extends Serializable {
```

Persistence Manager

```
PersistenceManager getPersistenceManager();
```

Return the associated `PersistenceManager` instance. If this `Query` instance was restored from a serialized form, then `null` is returned.

Query element binding

The `Query` interface provides methods to bind required and other elements prior to execution.

All of these methods replace the previously set query element, by the parameter. [The methods are not additive.] For example, if multiple variables are needed in the query, all of them must be specified in the same call to `declareVariables`.

```
void setClass (Class candidateClass);
```

Bind the candidate class to the query instance.

```
void setCandidates (Collection candidateCollection);
```

Bind the candidate `Collection` to the query instance. If the user adds or removes elements from the `Collection` after this call, it is not determined whether the added/removed elements take part in the `Query`, or whether a `NoSuchElementException` is thrown during execution of the `Query`.

For portability, the elements in the collection must be persistent instances associated with the same `PersistenceManager` as the `Query` instance. An implementation might support transient instances in the collection. If persistent instances associated with another `PersistenceManager` are in the collection, `JDOUserException` is thrown during `execute()`.

If the candidates are not specified explicitly by `newQuery`, `setCandidates(Collection)`, or `setCandidates(Extent)`, then the candidate extent is the extent of instances of the candidate class in the datastore including subclasses. That is, the candidates are the result of `getPersistenceManager().getExtent(candidateClass, true)`.

```
void setCandidates (Extent candidateExtent);
```

Bind the candidate `Extent` to the query instance.

```
void setFilter (String filter);
```

Bind the query filter to the query instance.

```
void declareImports (String imports);
```

Bind the import statements to the query instance. All imports must be declared in the same method call, and the imports must be separated by semicolons.

```
void declareVariables (String variables);
```

Bind the variable statements to the query instance. This method defines the types and names of variables that will be used in the filter but not provided as values by the `execute` method.

```
void declareParameters (String parameters);
```

Bind the parameter statements to the query instance. This method defines the parameter types and names that will be used by a subsequent `execute` method.

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

Bind the ordering statements to the query instance.

Query options

```
void setIgnoreCache (boolean flag);
```

```
boolean getIgnoreCache ();
```

The `IgnoreCache` option, when set to `true`, is a hint to the query engine that the user expects queries be optimized to return approximate results by ignoring changed values in the cache. This option is useful only for optimistic transactions and allows the datastore to return results that do not take modified cached instances into account. An implementation

may choose to ignore the setting of this flag, and always return exact results reflecting current cached values, as if the value of the flag were `false`.

Query compilation

The `Query` interface provides a method to compile queries for subsequent execution.

```
void compile();
```

This method requires the `Query` instance to validate any elements bound to the query instance and report any inconsistencies by throwing a `JDOUserException`. It is a hint to the `Query` instance to prepare and optimize an execution plan for the query.

14.6.1 Query execution

The `Query` interface provides methods that execute the query based on the parameters given. They return an unmodifiable `Collection` which the user can iterate to get results. Executing any operation on the `Collection` that might change it throws `UnsupportedOperationException`. For future extension, the signature of the `execute` methods specifies that they return an `Object` that must be cast to `Collection` by the user.

Any parameters passed to the `execute` methods are used only for this execution, and are not remembered for future execution.

For portability, parameters of persistence-capable types must be persistent or transactional instances. Parameters that are persistent or transactional instances must be associated with the same `PersistenceManager` as the `Query` instance. An implementation might support transient instances of persistence-capable types as parameters. If a persistent instance associated with another `PersistenceManager` is passed as a parameter, `JDOUserException` is thrown during `execute()`.

Queries may be constructed at any time before the `PersistenceManager` is closed, but may be executed only at certain times. If the `PersistenceManager` that constructed the `Query` is closed, then the `execute` methods throw `JDOUserException`. If the `Non-transactionalRead` property is `false`, and a transaction is not active, then the `execute` methods throw `JDOUserException`.

```
Object execute();
```

```
Object execute(Object p1);
```

```
Object execute(Object p1, Object p2);
```

```
Object execute(Object p1, Object p2, Object p3);
```

The `execute` methods execute the query using the parameters and return the result, which is an unmodifiable collection of instances that satisfy the boolean filter. The result may be a large `Collection`, which should be iterated or possibly passed to another `Query`. The `size()` method might return `Integer.MAX_VALUE` if the actual size of the result is not known (for example, the `Collection` represents a censored result).

When using an `Extent` to define candidate instances, the contents of the extent are subject to the setting of the `ignoreCache` flag. With `ignoreCache` set to `false`:

- if instances were made persistent in the current transaction, the instances will be considered part of the candidate instances.
- if instances were deleted in the current transaction, the instances will not be considered part of the candidate instances.

With `ignoreCache` set to `true`:

- if instances were made persistent in the current transaction, the new instances might not be considered part of the candidate instances.
- if instances were deleted in the current transaction, the instances will not be considered part of the candidate instances.

Each parameter of the `execute` method(s) is an `Object` that is either the value of the corresponding parameter or the wrapped value of a primitive parameter. The parameters associate in order with the parameter declarations in the `Query` instance.

```
Object executeWithMap (Map m);
```

The `executeWithMap` method is similar to the `execute` method, but takes its parameters from a `Map` instance. The `Map` contains key/value pairs, in which the key is the declared parameter name, and the value is the value to use in the query for that parameter. Unlike `execute`, there is no limit on the number of parameters.

```
Object executeWithArray (Object[] a);
```

The `executeWithArray` method is similar to the `execute` method, but takes its parameters from an array instance. The array contains `Objects`, in which the positional `Object` is the value to use in the query for that parameter. Unlike `execute`, there is no limit on the number of parameters.

14.6.2 Filter specification

The filter specification is a `String` containing a boolean expression that is to be evaluated for each of the instances in the candidate collection. If the filter is not specified, then it defaults to "true", and the input `Collection` is filtered only for class type.

An element of the candidate collection is returned in the result if:

- it is assignment compatible to the candidate `Class` of the `Query`; and
- for all variables there exists a value for which the filter expression evaluates to true. The user may denote uniqueness in the filter expression by explicitly declaring an expression (for example, `e1 != e2`). For example, a filter for a `Department` where there exists an `Employee` with more than one dependent and an `Employee` making more than 30,000 might be: `"(emps.contains(e1) & e1.dependents > 1) & (emps.contains(e2) & e2.salary > 30000)"`. The same `Employee` might satisfy both conditions. But if the query required that there be two different `Employees` satisfying the two conditions, an additional expression could be added: `"(emps.contains(e1) & e1.dependents > 1) & (emps.contains(e2) & (e2.salary > 30000 & e1 != e2))"`.

Rules for constructing valid expressions follow the Java language, except for these differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of `Date` fields and `Date` parameters are valid.
- Equality and ordering comparisons of `String` fields and `String` parameters are valid. The comparison is done according to an ordering not specified by JDO. This allows an implementation to order according to a datastore-specified ordering, which might be locale-specific.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.

- The assignment operators `=`, `+=`, etc. and pre- and post-increment and -decrement are not supported.
- Methods, including object construction, are not supported, except for `Collection.contains(Object o)`, `Collection.isEmpty()`, `String.startsWith(String s)`, and `String.endsWith(String e)`. Implementations might choose to support non-mutating method calls as non-standard extensions.
- Navigation through a null-valued field, which would throw `NullPointerException`, is treated as if the subexpression returned `false`. Similarly, a failed cast operation, which would throw `ClassCastException`, is treated as if the subexpression returned `false`. Other subexpressions or other values for variables might still qualify the candidate instance for inclusion in the result set.
- Navigation through multi-valued fields (`Collection` types) is specified using a variable declaration and the `Collection.contains(Object o)` method.
- The following literals are supported, as described in the Java Language Specification: `IntegerLiteral`, `FloatingPointLiteral`, `BooleanLiteral`, `CharacterLiteral`, `StringLiteral`, and `NullLiteral`.

Note that comparisons between floating point values are by nature inexact. Therefore, equality comparisons (`==` and `!=`) with floating point values should be used with caution.

Identifiers in the expression are considered to be in the name space of the specified class, with the addition of declared imports, parameters and variables. As in the Java language, `this` is a reserved word, and it refers to the element of the collection being evaluated.

Identifiers that are persistent field names are required to be supported by JDO implementations. Identifiers that are not persistent field names (including final and static field names) might be supported but are not required. Portable queries must not use non-persistent, final, or static field names in filter expressions.

Navigation through single-valued fields is specified by the Java language syntax of `field_name.field_name....field_name`.

A JDO implementation is allowed to reorder the filter expression for optimization purposes.

The following are minimum capabilities of the expressions that every implementation must support:

- operators applied to all types where they are defined in the Java language:

Table 4: Query Operators

Operator	Description
<code>==</code>	equal
<code>!=</code>	not equal
<code>></code>	greater than
<code><</code>	less than
<code>>=</code>	greater than or equal

Table 4: Query Operators

Operator	Description
<=	less than or equal
&	boolean logical AND (not bitwise)
&&	conditional AND
	boolean logical OR (not bitwise)
	conditional OR
~	integral unary bitwise complement
+	binary or unary addi- tion or String concate- nation
-	binary subtraction or numeric sign inversion
*	times
/	divide by
!	logical complement

- exceptions to the above:
 - String concatenation is supported only for `String + String`, not `String + <primitive>`;
- parentheses to explicitly mark operator precedence
- cast operator (class)
- promotion of numeric operands for comparisons and arithmetic operations. The rules for promotion follow the Java rules (see chapter 5.6 Numeric Promotions of the Java language spec) extended by `BigDecimal`, `BigInteger` and numeric wrapper classes:
 - if either operand is of type `BigDecimal`, the other is converted to `BigDecimal`.
 - otherwise, if either operand is of type `BigInteger`, and the other type is a floating point type (`float`, `double`) or one of its wrapper classes (`Float`, `Double`) both operands are converted to `BigDecimal`.
 - otherwise, if either operand is of type `BigInteger`, the other is converted to `BigInteger`.
 - otherwise, if either operand is of type `double`, the other is converted to `double`.
 - otherwise, if either operand is of type `float`, the other is converted to `float`.
 - otherwise, if either operand is of type `long`, the other is converted to `long`.
 - otherwise, both operands are converted to type `int`.

- operands of numeric wrapper classes are treated as their corresponding primitive types. If one of the operands is of a numeric wrapper class and the other operand is of a primitive numeric type, the rules above apply and the result is of the corresponding numeric wrapper class.
- equality comparison among persistent instances of `PersistenceCapable` types use the JDO Identity comparison of the references. Thus, two objects will compare equal if they have the same JDO Identity.
- comparisons between persistent and non-persistent instances return not equal.
- equality comparison of instances of non-`PersistenceCapable` reference types uses the `equals` method of the type.
- `String` methods `startsWith` and `endsWith` support wild card queries. JDO does not define any special semantic to the argument passed to the method; in particular, it does not define any wild card characters.
- Null-valued fields of `Collection` types are treated as if they were empty if a method is called on them. In particular, they return `true` to `isEmpty` and return `false` to all `contains` methods. For datastores that support null values for `Collection` types, it is valid to compare the field to null. Datastores that do not support null values for `Collection` types, will return `false` if the query compares the field to null. Datastores that support null values for `Collection` types should include the option `"javax.jdo.option.NullCollection"` in their list of supported options (`PersistenceManagerFactory.supportedOptions()`).

14.6.3 Parameter declaration

The parameter declaration is a `String` containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

Parameter types for primitive values can be specified as either the primitive types or the corresponding wrapper types. If a parameter type is specified as a primitive, the parameter value passed to `execute()` must not be null.

14.6.4 Import statements

The import statements follow the Java syntax for import statements.

14.6.5 Variable declaration

The type declarations follow the Java syntax for local variable declarations.

If the variable is not named in a `contains` clause, that variable's scope while evaluating the filter expression is the `Extent` (including subclasses) of the class of the variable. If the class does not manage an `Extent`, then no results will satisfy the query.

A portable query will constrain all variables with a `contains` clause in each "OR" expression of the filter where the variable is used. Further, the `contains` clause must be the left expression of an "AND" expression where the variable is used in the right expression. That is, for each occurrence of an expression in the filter using the variable, there is a `contains` clause "ANDed" with the expression that constrains the possible values by the elements of a collection.

A variable that is not constrained with an explicit `contains` clause is constrained by the extent of the persistence capable class in the database.

The semantics of `contains` is "exists". The meaning of the expression `"emps.contains(e) && e.salary < param"` is "there exists an `e` in the `emps` collection such that `e.salary` is less than

param”. This is the natural meaning of contains in the Java language, except where the expression is negated.

If the expression is negated, then “!(emps.contains(e) && e.salary < param)” means “there does not exist an employee e in the collection emps such that e.salary is less than param”. Another way of expressing this is “for each employee e in the collection emps, e.salary is greater than or equal to param”.

14.6.6 Ordering statement

The ordering statement is a `String` containing one or more ordering declarations separated by commas. Each ordering declaration is a Java expression of an orderable type:

- primitives except `boolean`;
- wrappers except `Boolean`;
- `BigDecimal`;
- `BigInteger`;
- `String`;
- `Date`

followed by one of the following words: “ascending” or “descending”.

Ordering might be specified including navigation. The name of the field to be used in ordering via navigation through single-valued fields is specified by the Java language syntax of `field_name.field_name...field_name`.

The result of the first (leftmost) expression is used to order the results. If the leftmost expression evaluates the same for two or more elements, then the second expression is used for ordering those elements. If the second expression evaluates the same, then the third expression is used, and so on until the last expression is evaluated. If all of the ordering expressions evaluate the same, then the ordering of those elements is unspecified.

The ordering of instances containing null-valued fields specified by the ordering is not specified. Different JDO implementations might order the instances containing null-valued fields either before or after instances whose fields contain non-null values.

14.6.7 Closing Query results

When the application has finished with the query results, it might optionally close the results, allowing the JDO implementation to release resources that might be engaged, such as database cursors or iterators. The following methods allow early release of these resources.

```
void close (Object queryResult);
```

This method closes the result of one `execute(...)` method, and releases resources associated with it. After this method completes, the query result can no longer be used, for example to iterate the returned elements. Any elements returned previously by iteration of the results remain in their current state. Any iterators acquired from the queryResult will return `false` to `hasNext()` and will throw `NoSuchElementException` to `next()`.

```
void closeAll ();
```

This method closes all results of `execute(...)` methods on this `Query` instance, as above. The `Query` instance is still valid and can still be used.

14.7 Examples:

The following class definitions for persistence capable classes are used in the examples:

```
package com.xyz.hr;
class Employee {
    String name;
    Float salary;
    Department dept;
    Employee boss;
}
package com.xyz.hr;
class Department {
    String name;
    Collection emps;
}
```

14.7.1 Basic query.

This query selects all Employee instances from the candidate collection where the salary is greater than the constant 30000.

Note that the float value for salary is unwrapped for the comparison with the literal int value, which is promoted to float using numeric promotion. If the value for the salary field in a candidate instance is null, then it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

14.7.2 Basic query with ordering.

This query selects all Employee instances from the candidate collection where the salary is greater than the constant 30000, and returns a Collection ordered based on employee salary.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000";
Query q = pm.newQuery (clnEmployee, filter);
q.setOrdering ("salary ascending");
Collection emps = (Collection) q.execute ();
```

14.7.3 Parameter passing.

This query selects all Employee instances from the candidate collection where the salary is greater than the value passed as a parameter.

If the value for the salary field in a candidate instance is null, then it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > sal";
Query q = pm.newQuery (clnEmployee, filter);
String param = "Float sal";
q.declareParameters (param);
Collection emps = (Collection) q.execute (new Float (30000.));
```

14.7.4 Navigation through single-valued field.

This query selects all Employee instances from the candidate collection where the value of the name field in the Department instance associated with the Employee instance is equal to the value passed as a parameter.

If the value for the dept field in a candidate instance is null, then it cannot be navigated for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "dept.name == dep";
String param = "String dep";
Query q = pm.newQuery (clnEmployee, filter);
q.declareParameters (param);
String rnd = "R&D";
Collection emps = (Collection) q.execute (rnd);
```

14.7.5 Navigation through multi-valued field.

This query selects all Department instances from the candidate collection where the collection of Employee instances contains at least one Employee instance having a salary greater than the value passed as a parameter.

```
Class depClass = Department.class;
Extent clnDepartment = pm.getExtent (depClass, false);
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
String param = "float sal";
Query q = pm.newQuery (clnDepartment, filter);
q.declareParameters (param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

14.7.6 Membership in a collection

This query selects all Department instances where the name field is contained in a parameter collection, which in this example consists of three department names.

```
Class depClass = Department.class;
Extent clnDepartment = pm.getExtent (depClass, false);
String filter = "depts.contains(name)";
List depts =
    Arrays.asList(new String [] {"R&D", "Sales", "Marketing"};
String param = "Collection depts";
Query q = pm.newQuery (clnDepartment, filter);
q.declareParameters (param);
Collection deps = (Collection) q.execute (depts);
```

15 Extent

This chapter specifies the `Extent` contract between an application component and the JDO implementation.

15.1 Overview

An application needs to provide a candidate collection of instances to a query. If the query filtering is to be performed in the datastore, then the application must supply the collection of instances to be filtered. This is the primary function of the `Extent` interface.

An `Extent` instance is logically a holder for information:

- the class of instances;
- whether subclasses are part of the `Extent`; and
- a collection of active iterators over the `Extent`.

Thus, no action is taken at the time the `Extent` is constructed. The contents of the `Extent` are calculated at the point in time when a query is executed and when an iterator is obtained via the `iterator()` method.

A query may be executed against either a `Collection` or an `Extent`. The `Extent` is used when the query is intended to be filtered by the datastore, not by in-memory processing. There are no `Collection` methods in `Extent` except for `iterator()`. Thus, common `Collection` behaviors are not possible, including determining whether one `Extent` contains another, determining the size of the `Extent`, or determining whether a specific instance is contained in the `Extent`. Any such operations must be performed by executing a query against the `Extent`.

If the `Extent` is large, then an appropriate iteration strategy should be adopted by the JDO implementation.

The `Extent` for classes of embedded instances is not affected by changes to fields in referencing class instances.

15.2 Goals

The extent interface has the following goals:

- Large result set support. Queries might return massive numbers of JDO instances that match the query. The JDO Query architecture must provide for processing the results within the resource constraints of the execution environment.
- Application resource management. Iterating an `Extent` might use resources that should be released when the application has finished an iteration. The application should be provided with a means to release iterator resources.

15.3 Interface Extent

```
package javax.jdo;  
  
public interface Extent {  
    Iterator iterator();
```

This method returns an `Iterator` over all the instances in the `Extent`. If `NontransactionalRead` property is set to `false`, this method will throw a `JDOUserException` if called outside a transaction.

If the `IgnoreCache` option is set to `true` in the `PersistenceManager` at the time that this `Iterator` instance is obtained, then new and deleted instances in the current transaction might be ignored by the `Iterator` at the option of the implementation. That is, new instances might not be returned; and deleted instances might be returned.

If the `IgnoreCache` option is set to `false` in the `PersistenceManager` at the time that this `Iterator` instance is obtained, then:

- If instances were made persistent in the transaction prior to the execution of this method, the returned `Iterator` will contain the instances.
- If instances were deleted in the transaction prior to the execution of this method, the returned `Iterator` will not contain the instances.

The above describes the behavior of an extent-based query at query execution.

If any mutating method, including the `remove` method, is called on the `Iterator` returned by this method, a `UnsupportedOperationException` is thrown.

```
boolean hasSubclasses();
```

This method returns an indicator of whether the extent is proper or includes subclasses.

```
Class getCandidateClass();
```

This method returns the class of the instances contained in it.

```
PersistenceManager getPersistenceManager();
```

This method returns the `PersistenceManager` that created it.

```
void close(Iterator i);
```

This method closes an `Iterator` acquired from this `Extent`. After this call, the parameter `Iterator` will return `false` to `hasNext()`, and will throw `NoSuchElementException` to `next()`. The `Extent` itself can still be used to acquire other iterators and can be used as the `Extent` for queries.

```
void closeAll ();
```

This method closes all iterators acquired from this `Extent`. After this call, all iterators acquired from this `Extent` will return `false` to `hasNext()`, and will throw `NoSuchElementException` to `next()`.

Readers primarily interested in JDO as a local persistence mechanism can ignore the following chapter. Skip to 17 – JDO Exceptions.

16 Enterprise Java Beans

Enterprise Java Beans (EJB) is a component architecture for development and deployment of distributed business applications. Java Data Objects is a suitable component for integration with EJB in these scenarios:

- Session Beans with JDO persistence-capable classes used to implement dependent objects;
- Entity Beans with JDO persistence-capable classes used as delegates for both Bean Managed Persistence and Container Managed Persistence.

16.1 Session Beans

A session bean should be associated with an instance of `PersistenceManagerFactory` that is established during a session life cycle event, and each business method should use an instance of `PersistenceManager` obtained from the `PersistenceManagerFactory`. The timing of when the `PersistenceManager` is obtained will vary based on the type of bean.

The bean class should contain instance variables that hold the associated `PersistenceManager` and `PersistenceManagerFactory`.

During activation of the bean, the `PersistenceManagerFactory` should be found via JNDI lookup. The `PersistenceManagerFactory` should be the same instance for all beans sharing the same datastore resource. This allows for the `PersistenceManagerFactory` to manage an association between the distributed transaction and the `PersistenceManager`.

When appropriate during the bean life cycle, the `PersistenceManager` should be acquired by a call to the `PersistenceManagerFactory`. The `PersistenceManagerFactory` should look up the transaction association of the caller, and return a `PersistenceManager` with the same transaction association. If there is no `PersistenceManager` currently enlisted in the caller's transaction, a new `PersistenceManager` should be created and associated with the transaction. The `PersistenceManager` should be registered for synchronization callbacks with the `TransactionManager`. This provides for transaction completion callbacks asynchronous to the bean life cycle.

The instance variables for a session bean of any type include:

- a reference to the `PersistenceManagerFactory`, which should be initialized by the method `setSessionContext`. This method looks up the `PersistenceManagerFactory` by JNDI access to the named object `"java:comp/env/jdo/<persistence manager factory name>"`.
- a reference to the `PersistenceManager`, which should be acquired by each business method, and closed at the end of the business method; and

- a reference to the `SessionContext`, which should be initialized by the method `setSessionContext`.

16.1.1 Stateless Session Bean with Container Managed Transactions

Stateless session beans are service objects that have no state between business methods. They are created as needed by the container and are not associated with any one user. A business method invocation on a remote reference to a stateless session bean might be dispatched by the container to any of the available beans in the ready pool.

Each business method must acquire its own `PersistenceManager` instance from the `PersistenceManagerFactory`. This is done via the method `getPersistenceManager` on the `PersistenceManagerFactory` instance. This method must be implemented by the JDO vendor to find a `PersistenceManager` associated with the instance of `javax.transaction.Transaction` of the executing thread.

At the end of the business method, the `PersistenceManager` instance must be closed. This allows the transaction completion code in the `PersistenceManager` to free the instance and return it to the available pool in the `PersistenceManagerFactory`.

16.1.2 Stateful Session Bean with Container Managed Transactions

Stateful session beans are service objects that are created for a particular user, and may have state between business methods. A business method invocation on a remote reference to a stateful session bean will be dispatched to the specific instance created by the user.

The behavior of stateful session beans with container managed transactions is otherwise the same as for stateless session beans. All business methods in the remote interface must acquire a `PersistenceManager` at the beginning of the method, and close it at the end, since the transaction context is managed by the container.

16.1.3 Stateless Session Bean with Bean Managed Transactions

Bean managed transactions offer additional flexibility to the session bean developer, with additional complexity. Transaction boundaries are established by the bean developer, but the state (including the `PersistenceManager`) cannot be retained across business method boundaries. Therefore, the `PersistenceManager` must be acquired and closed by each business method.

The alternative techniques for transaction boundary demarcation are:

- `javax.transaction.UserTransaction`

If the bean developer directly uses `UserTransaction`, then the `PersistenceManager` must be acquired from the `PersistenceManagerFactory` only after establishing the correct transaction context of `UserTransaction`. During the `getPersistenceManager` method, the `PersistenceManager` will be enlisted in the `UserTransaction`. For example, if non-transactional access is required, a `PersistenceManager` must be acquired when there is no `UserTransaction` active. After beginning a `UserTransaction`, a different `PersistenceManager` must be acquired for transactional access. The user must keep track of which `PersistenceManager` is being used for which transaction.

- `javax.jdo.Transaction`

If the bean developer chooses to use the same `PersistenceManager` for multiple transactions, then transaction completion must be done entirely by using the `jav-`

`ax.jdo.Transaction` instance associated with the `PersistenceManager`. In this case, acquiring a `PersistenceManager` without beginning a `UserTransaction` results in the `PersistenceManager` being able to manage transaction boundaries via `begin`, `commit`, and `rollback` methods on `javax.jdo.Transaction`. The `PersistenceManager` will automatically begin the `UserTransaction` during `javax.jdo.Transaction.begin` and automatically commit the `UserTransaction` during `javax.jdo.Transaction.commit`.

16.1.4 Stateful Session Bean with Bean Managed Transactions

Stateful session beans allow the bean developer to manage the transaction context as part of the conversational state of the bean. Thus, it is no longer required to acquire a `PersistenceManager` in each business method. Instead, the `PersistenceManager` can be managed over a longer period of time, and it might be stored as an instance variable of the bean.

The behavior of stateful session beans is otherwise the same as for stateless session beans. The user has the choice of using `javax.transaction.UserTransaction` or `javax.jdo.Transaction` for transaction completion.

16.2 Entity Beans

There are several components in the entity bean scenario that need to be considered to understand the JDO integration possibilities.

EJBObject: this is the class of non-transactional instances to which all remote proxies make reference so they can execute remote calls. When the EJB container receives a remote call on this instance, it looks up a transactional instance of the Entity Bean associated with the user's transaction and delegates the business method to it.

Entity Bean: this is the class of transactional instances whose life cycle is managed by the container to service remote method calls. At certain points in its life cycle an instance will be associated with an `EntityContext` instance, a `PersistenceManager` instance, and a JDO instance.

The Entity Bean might be constructed manually by the bean developer or automatically by a tool provided by a JDO vendor or third party.

JDO PersistenceCapable class: this is the class of instances that actually implement the business method by accessing and possibly modifying state. During execution of its business methods, it may associate with other JDO instances, which themselves are in the transaction but do not have remote references. If during the execution of a method, a reference to a JDO instance needs to be returned to the client, then the Entity Bean will ask the Home Interface to construct an `EJBObject` using the primary key of the referred JDO instance.

JDO instances might be used to implement both the Entity Bean and the helper instances that are transactional but not returned to clients as remote references.

16.2.1 BMP Entity Bean life cycle

The Entity Bean that delegates to a JDO instance will contain a reference to an `EntityContext` instance, a `PersistenceManager` instance, and a JDO instance. All of these will be set to `null` at creation time.

The `setEntityContext` method copies the value of the `EntityContext` parameter to the instance variable `entityContext`, looks up the `PersistenceManagerFactory` using JNDI, and sets it into the `persistenceManagerFactory` variable.

The `unsetEntityContext` method clears the `entityContext` variable and the `persistenceManagerFactory` variable.

The `ejbCreate` method gets the `PersistenceManager` from the `PersistenceManagerFactory`, creates a new instance of the corresponding JDO class corresponding to the primary key value, sets the `jdoInstance` variable, and calls `makePersistent` with the instance as a parameter.

The `ejbRemove` method calls `deletePersistent` with the JDO instance as a parameter.

The `ejbActivate` method does nothing.

The `ejbPassivate` method clears the reference to the `PersistenceManager` and JDO instance.

The `ejbLoad` method acquires the `PersistenceManager`. Business methods operating on the JDO instance will access fields which will transparently cause the appropriate state transitions of the persistent instance. The state of the JDO instances will be synchronized with the datastore during transaction completion.

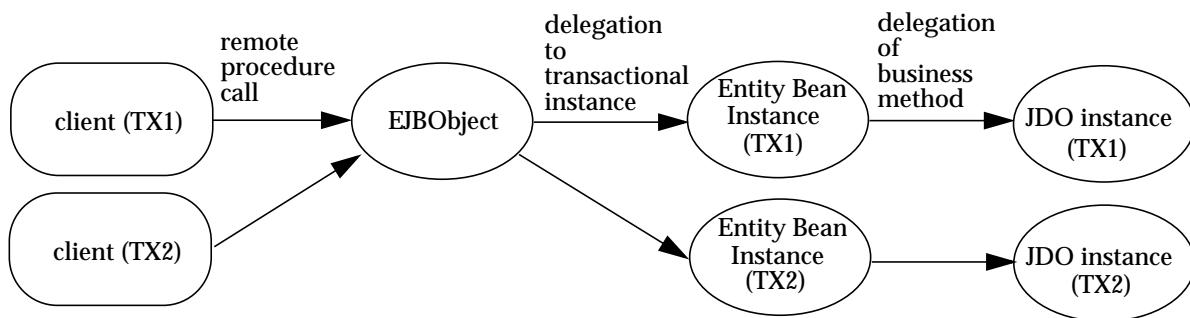
The `ejbStore` method does nothing.

Business methods of the bean delegate to the JDO instance after performing parameter replacement operations. Parameters of Entity Bean reference types are transformed into parameters of JDO instances by using `getObjectById`.

During execution of business methods, persistent instances may be used without regard to EJB. The persistent instances might be of classes that have remote interfaces, but there is no requirement that an `EJBObject` be created for each persistent instance. Only those instances that need to be returned as a remote interface need to have an `EJBObject` created.

JDO instances may be returned as value types using standard Java Serialization. The main consideration is that the closure of instances via non-transient fields be appropriate. Considering that the closure of instances via persistent fields is potentially the entire database, care must be given to distinguishing persistent fields from serializable fields.

JDO instance return types are transformed into Entity Bean references by using `getObjectId` and looking up the EJB instance in the home interface.

Figure 16.0 Enterprise Java Beans: Entity Bean relationships

17 JDO Exceptions

The exception philosophy of JDO is to treat all exceptions as runtime exceptions. This preserves the transparency of the interface to the degree possible, allowing the user to choose to catch specific exceptions only when required by the application.

JDO implementations will often be built as layers on an underlying datastore interface, which itself might use a layered protocol to another tier. Therefore, there are many opportunities for components to fail that are not under the control of the application.

Exceptions thus fall into several broad categories, each of which is treated separately:

- user errors that can be corrected and retried;
- user errors that cannot be corrected because the state of underlying components has been changed and cannot be undone;
- internal logic errors that should be reported to the JDO vendor's technical support;
- errors in the underlying datastore that can be corrected and retried;
- errors in the underlying datastore that cannot be corrected due to a failure of the datastore or communication path to the datastore;

Exceptions that are documented in interfaces that are used by JDO, such as the `Collection` interfaces, are used without modification by JDO. JDO exceptions that reflect underlying datastore exceptions will wrap the underlying datastore exceptions. JDO exceptions that are caused by user errors will contain the reason for the exception.

JDO Exceptions must be serializable.

17.1 JDOException

This is the base class for all JDO exceptions. It is a subclass of `RuntimeException`, and need not be declared or caught. It includes a descriptive String, an optional nested Exception array, and an optional failed Object.

Methods are provided to retrieve the nested exception array and failed object. If there are multiple nested exceptions, then each might contain one failed object. This will be the case where an operation requires multiple instances, such as `commit`, `makePersistentAll`, etc.

If the JDO `PersistenceManager` is internationalized, then the descriptive string should be internationalized.

```
public Throwable[] getNestedExceptions();
```

This method returns an array of `Throwable` or `null` if there are no nested exceptions.

```
public Object getFailedObject();
```

This method returns the failed object or `null` if there is no failed object for this exception.

```
public Throwable getCause();
```

This method returns the first nested `Throwable` or `null` if there are no nested exceptions.

17.1.1 JDOFatalException

This is the base class for errors that cannot be retried. It is a derived class of `JDOException`. This exception generally means that the transaction associated with the `PersistenceManager` has been rolled back, and the transaction should be abandoned.

17.1.2 JDOCanRetryException

This is the base class for errors that can be retried. It is a derived class of `JDOException`.

17.1.3 JDOUnsupportedOptionException

This class is a derived class of `JDOUserException`. This exception is thrown by an implementation to indicate that it does not implement a JDO optional feature.

17.1.4 JDOUserException

This is the base class for user errors that can be retried. It is a derived class of `JDOCanRetryException`. Some of the reasons for this exception include:

- **Object not `PersistenceCapable`.** This exception is thrown when a method requires an instance of `PersistenceCapable` and the instance passed to the method does not implement `PersistenceCapable`. The failed Object has the failed instance.
- **Extent not managed.** This exception is thrown when `getExtent` is called with a class that does not have a managed extent.
- **Object exists.** This exception is thrown during flush of a new instance or an instance whose primary key changed where the primary key of the instance already exists in the datastore. It might also be thrown during `makePersistent` if an instance with the same primary key is already in the `PersistenceManager` cache. The failed Object is the failed instance.
- **Object owned by another `PersistenceManager`.** This exception is thrown when calling `makePersistent`, `makeTransactional`, `makeTransient`, `evict`, `refresh`, or `getObjectId` where the instance is already persistent or transactional in a different `PersistenceManager`. The failed Object has the failed instance.
- **Non-unique `ObjectId` not valid after transaction completion.** This exception is thrown when calling `getObjectId` on an object after transaction completion where the `ObjectId` is not managed by the application or datastore.
- **Unbound query parameter.** This exception is thrown during query compilation or execution if there is an unbound query parameter.
- **Query filter cannot be parsed.** This exception is thrown during query compilation or execution if the filter cannot be parsed.
- **Transaction is not active.** This exception is thrown if the transaction is not active and `makePersistent`, `deletePersistent`, `commit`, or `rollback` is called.
- **Object deleted.** This exception is thrown if an attempt is made to access any fields of an instance that was deleted in this transaction (except to read key fields). This is not the exception thrown if the instance does not exist in the datastore (see `JDOObjectNotFoundException`).
- **Primary key contains null values.** This exception is thrown if the application identity parameter to `getObjectById` contains any key field whose value is null.

17.1.5 JDOFatalUserException

This is the base class for user errors that cannot be retried. It is a derived class of `JDOFatalException`.

- `PersistenceManager` was closed. This exception is thrown after `close()` was called, when any method except `isClosed()` is executed on the `PersistenceManager` instance, or any method is called on the `Transaction` instance, or any `Query` instance, `Extent` instance, or `Iterator` instance created by the `PersistenceManager`.
- Metadata unavailable. This exception is thrown if a request is made to the `JDOImplHelper` for metadata for a class, when the class has not been registered with the helper.

17.1.6 JDOFatalInternalException

This is the base class for JDO implementation failures. It is a derived class of `JDOFatalException`. This exception should be reported to the vendor for corrective action. There is no user action to recover.

17.1.7 JDODataStoreException

This is the base class for datastore errors that can be retried. It is a derived class of `JDOCanRetryException`.

17.1.8 JDOFatalDataStoreException

This is the base class for fatal datastore errors. It is a derived class of `JDOFatalException`. When this exception is thrown, the transaction has been rolled back.

- Transaction rolled back. This exception is thrown when the datastore rolls back a transaction without the user asking for it. The cause may be a connection timeout, an unrecoverable media error, an unrecoverable concurrency conflict, or other cause outside the user's control.

17.1.9 JDOObjectNotFoundException

This exception is to notify the application that an object does not exist in the datastore. It is a derived class of `JDODataStoreException`. When this exception is thrown during a transaction, there has been no change in the status of the transaction in progress. If this exception is a nested exception thrown during commit, then the transaction is rolled back. This exception is never the result of executing a query. The `failedObject` contains a reference to the failed instance. The failed instance is in the hollow state, and has an identity which can be obtained by calling `getObjectId` with the instance as a parameter. This might be used to determine the identity of the instance that cannot be found.

This exception is thrown when a hollow instance is being fetched and the object does not exist in the datastore. This exception might result from the user executing `getObjectById` with the `validate` parameter set to `true`, or from navigating to an object that no longer exists in the datastore.

17.1.10 JDOOptimisticVerificationException

This exception is the result of a user commit operation in an optimistic transaction where the verification of new, modified, or deleted instances fails the verification. It is a derived class of `JDOFatalDataStoreException`. This exception contains an array of nested exceptions, each of which contains an instance that failed verification. The user will never see this exception except as a result of commit.

18 XML Metadata

This chapter specifies the metadata that describes a persistence-capable class. The metadata is stored in XML format. The information must be available when the class is enhanced, and might be cached by an implementation for use at runtime. If the metadata is changed between enhancement and runtime, the behavior is unspecified.

Metadata files must be available via resources loaded by the same class loader as the class. These rules apply both to enhancement and to runtime. Hereinafter, the term "metadata" refers to the aggregate of all XML data for all packages and classes, regardless of their physical packaging.

The metadata associated with each persistence capable class must be contained within a file, and its format is defined by the DTD. If the metadata is for only one class, then its file name is `<class-name>.jdo`. If the metadata is for a package, or a number of packages, then its file name is `package.jdo`. In this case, the file is located in one of several directories: "META-INF"; "WEB-INF"; `<none>`, in which case the metadata file name is "package.jdo" with no directory; `<package>/.../<package>`, in which case the metadata directory name is the partial or full package name with "package.jdo" as the file name.

When metadata information is needed for a class, and the metadata for that class has not already been loaded, the metadata is searched as follows: META-INF/package.jdo, WEB-INF/package.jdo, package.jdo, `<package>/.../<package>/package.jdo`, and `<package>/<class>.jdo`. Once metadata for a class has been loaded, the metadata will not be replaced in memory. Therefore, metadata contained higher in the search order will always be used instead of metadata contained lower in the search order.

For example, if the persistence-capable class is `com.xyz.Wombat`, and there is a file "META-INF/package.jdo" containing xml for this class, then its definition is used. If there is no such file, but there is a file "WEB-INF/package.jdo" containing metadata for `com.xyz.Wombat`, then it is used. If there is no such file, but there is a file "package.jdo" containing metadata for `com.xyz.Wombat`, then it is used. If there is no such file, but there is a file "com/package.jdo" containing metadata for `com.xyz.Wombat`, then it is used. If there is no such file, but there is a file "com/xyz/package.jdo" containing metadata for `com.xyz.Wombat`, then it is used. If there is no such file, but there is a file "com/xyz/Wombat.jdo", then it is used. If there is no such file, then `com.xyz.Wombat` is not persistence-capable.

Note that this search order is optimized for implementations that cache metadata information as soon as it is encountered so as to optimize the number of file accesses needed to load the metadata. Further, if metadata is not in the natural location, it might override metadata that is in the natural location. For example, while looking for metadata for class `com.xyz.Wombat`, the file `com/package.jdo` might contain metadata for class `org.acme.Foo`. In this case, subsequent search of metadata for `org.acme.Foo` will find the cached metadata and none of the usual locations for metadata will be searched.

The metadata must declare all persistence-capable classes. If any field declarations are not provided in the metadata, then field metadata is defaulted for the missing field declarations. Therefore, the JDO implementation is able to determine based on the metadata

whether a class is persistence-capable or not. And any class not known to be persistence-capable by the JDO specification (for example, `java.lang.Integer`) and not explicitly named in the metadata is not persistence-capable.

For compatibility with installed applications, an implementation might first use the search order as specified in the JDO 1.0 release. In this case, if metadata is not found, then the search order as specified in JDO 1.0.1 must be used.

18.1 ELEMENT `jdo`

This element is the highest level element in the xml document. It is used to allow multiple packages to be described in the same document.

18.2 ELEMENT `package`

This element includes all classes in a particular package. The complete qualified package name is required.

18.3 ELEMENT `class`

This element includes fields declared in a particular class, and optional vendor extensions. The name of the class is required. The name is relative to the package name of the enclosing package.

Only persistence-capable classes may be declared. Non-persistence-capable classes must not be included in the metadata.

The identity type of the least-derived persistence-capable class defines the identity type for all persistence-capable classes that extend it.

The identity type of the least-derived persistence-capable class is defaulted to `application` if `objectid-class` is specified, and `datastore`, if not.

The `objectid-class` attribute is required only for application identity. The `objectid` class name uses Java rules for naming: if no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the “\$” marker. If the `objectid-class` attribute is defined in any concrete class, then the `objectid` class itself must be concrete, and no subclass of the class may include the `objectid-class` attribute. If the `objectid-class` attribute is defined for any abstract class, then:

- the `objectid` class of this class must directly inherit `Object` or must be a subclass of the `objectid` class of the most immediate abstract persistence-capable superclass that defines an `objectid` class; and
- if the `objectid` class is abstract, the `objectid` class of this class must be a superclass of the `objectid` class of the most immediate subclasses that define an `objectid` class; and
- if the `objectid` class is concrete, no subclass of this persistence-capable class may define an `objectid` class.

The effect of this is that `objectid` classes form an inheritance hierarchy corresponding to the inheritance hierarchy of the persistence-capable classes. Associated with every concrete persistence-capable class is exactly one `objectid` class.

The `objectid` class must declare fields identical in name and type to fields declared in this class.

The `requires-extent` attribute specifies whether an extent must be managed for this class. The `PersistenceManager.getExtent` method can be executed only for classes whose metadata attribute `requires-extent` is specified or defaults to `true`. If the `PersistenceManager.getExtent` method is executed for a class whose metadata specifies `requires-extent` as `false`, a `JDOUserException` is thrown. If `requires-extent` is specified or defaults to `true` for a class, then `requires-extent` must not be specified as `false` for any subclass.

The `persistence-capable-superclass` attribute is the class name of the nearest superclass that is persistence-capable, and is required only for classes that have a persistence-capable superclass. The `persistence-capable-superclass` class name uses Java rules for naming; if no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. If omitted, there is no superclass in the hierarchy that is persistence-capable.

18.4 ELEMENT field

The `element` field is optional, and the `name` attribute is the field name as declared in the class. If the field declaration is omitted in the xml, then the values of the attributes are defaulted.

The `persistence-modifier` attribute specifies whether this field is persistent, transactional, or none of these. The `persistence-modifier` attribute can be specified only for fields declared in the Java class, and not fields inherited from superclasses. There is special treatment for fields whose `persistence-modifier` is persistent or transactional.

Default persistence-modifier

The default for the `persistence-modifier` attribute is based on the Java type and modifiers of the field:

- Fields with modifier `static`: none. No accessors or mutators will be generated for these fields during enhancement.
- Fields with modifier `transient`: none. Accessors and mutators will be generated for these fields during enhancement, but they will not delegate to the `StateManager`.
- Fields with modifier `final`: none. Accessors will be generated for these fields during enhancement, but they will not delegate to the `StateManager`.
- Fields of a type declared to be persistence-capable: persistent.
- Fields of the following types: persistent:
 - primitives: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`;
 - `java.lang` wrappers: `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, `Double`;
 - `java.lang`: `String`, `Number`;
 - `java.math`: `BigDecimal`, `BigInteger`;
 - `java.util`: `Date`, `Locale`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, `Vector`, `Collection`, `Set`, `List`, and `Map`;

- Arrays of primitive types, `java.util.Date`, `java.util.Locale`, `java.lang` and `java.math` types specified immediately above, and `PersistenceCapable` types.
- Fields of types of user-defined classes and interfaces not mentioned above: none. No accessors or mutators will be generated for these fields.

The `primary-key` attribute is used to identify fields that have special treatment by the enhancer and by the runtime. The enhancer generates accessor methods for primary key fields that always permit access, regardless of the state of the instance. The mutator methods always delegate to the `jdoStateManager`, if it is non-null, regardless of the state of the instance.

The `null-value` attribute specifies the treatment of null values for persistent fields during storage in the datastore. The default is "none".

- "none": store null values as null in the datastore, and throw a `JDOUserException` if null values cannot be stored by the datastore.
- "exception": always throw a `JDOUserException` if this field contains a null value at runtime when the instance must be stored;
- "default": convert the value to the datastore default value if this field contains a null value at runtime when the instance must be stored.

The `default-fetch-group` attribute specifies whether this field is managed as a group with other fields. It defaults to "true" for non-key fields of primitive types, `java.util.Date`, and fields of `java.lang`, `java.math` types specified above.

The `embedded` attribute specifies whether the field should be stored as part of the containing instance instead of as its own instance in the datastore. It must be specified or default to "true" for fields of primitive types, wrappers, `java.lang`, `java.math`, `java.util`, collection, map, and array types specified above; and "false" otherwise. While a compliant implementation is permitted to support these types as first class instances in the datastore, the semantics of `embedded="true"` imply containment. That is, the embedded instances have no independent existence in the datastore and have no `Extent` representation.

If the `embedded` attribute is "true" the field values are stored as persistent references to the referred instances in the datastore.

The `embedded` attribute applied to a field of a `PersistenceCapable` type is a hint to the implementation to treat the field as if it were a Second Class Object. But this behavior is not further specified and is not portable.

A portable application must not assign instances of mutable classes to multiple embedded fields, and must not compare values of these fields using Java identity ("`f1==f2`").

The following field declarations are mutually exclusive; only one may be specified:

- `default-fetch-group = "true"`
- `primary-key = "true"`
- `persistence-modifier = "transactional"`
- `persistence-modifier = "none"`

18.4.1 ELEMENT collection

This element specifies the element type of collection typed fields. The default is `Collection` typed fields are persistent, and the element type is `Object`.

The `element-type` attribute specifies the type of the elements. The type name uses Java rules for naming: if no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the "\$" marker.

The `embedded-element` attribute specifies whether the values of the elements should be stored as part of the containing instance instead of as their own instances in the datastore. It defaults to "false" for `PersistenceCapable` types, `Object` types, and interface types; and "true" for other types.

The embedded treatment of the collection instance itself is governed by the `embedded-attribute` of the `field` element.

18.4.2 ELEMENT map

This element specifies the treatment of keys and values of map typed fields. The default is map typed fields are persistent, and the key and value types are `Object`.

The `key-type` and `value-type` attributes specify the types of the key and value, respectively. The type names use Java rules for naming: if no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the "\$" marker.

The `embedded-key` and `embedded-value` attributes specify whether the key and value should be stored as part of the containing instance instead of as their own instances in the datastore. They default to "false" for `PersistenceCapable` types, `Object` types, and interface types; and "true" for other types.

The embedded treatment of the map instance itself is governed by the `embedded-attribute` of the `field` element.

18.4.3 ELEMENT array

This element specifies the treatment of array typed fields. The default persistence-modifier for array typed fields is based on the Java type of the component and modifiers of the field, according to the rules in **18.4 Default persistence-modifier**.

The `embedded-element` attribute specifies whether the values of the components should be stored as part of the containing instance instead of as their own instances in the datastore. It defaults to "false" for `PersistenceCapable` types, `Object` types, interface types, and concrete implementation classes of map and collection types. It defaults to "true" for other types.

The embedded treatment of the array instance itself is governed by the `embedded-attribute` of the `field` element.

18.5 ELEMENT extension

This element specifies JDO vendor extensions. The `vendor-name` attribute is required. The vendor name "JDORI" is reserved for use by the JDO reference implementation. The `key` and `value` attributes are optional, and have vendor-specific meanings. They may be ignored by any JDO implementation.

18.6 The Document Type Descriptor

The document type descriptor is referred by the xml, and must be identified with a DOCTYPE so that the parser can validate the syntax of the metadata file. Either the SYSTEM or PUBLIC form of DOCTYPE can be used.

- If SYSTEM is used, the URI must be accessible; a jdo implementation might optimize access for the URI "file:/javax/jdo/jdo.dtd"
- If PUBLIC is used, the public id should be "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"; a jdo implementation might optimize access for this id.

```
<?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">
<!ELEMENT jdo ((package)+, (extension)*)>
<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>
<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|nondurable)
    #IMPLIED>
<!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) #IMPLIED>
<!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 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 array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

18.7 Example XML file

An example XML file for the query example classes follows. Note that all fields of both classes are persistent, which is the default for fields. The `emps` field in `Department` contains a collection of elements of type `Employee`, with an inverse relationship to the `dept` field in `Employee`.

In directory `com/xyz`, a file named `hr.jdo` contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
<package name="com.xyz.hr">
<class name="Employee" identity-type="application" objectid-
class="EmployeeKey">
<field name="name" primary-key="true">
<extension vendor-name="sunw" key="index" value="btree"/>
</field>
<field name="salary" default-fetch-group="true"/>
<field name="dept">
<extension vendor-name="sunw" key="inverse" value="emps"/>
</field>
<field name="boss"/>
</class>
<class name="Department" identity-type="application" objectid-
class="DepartmentKey">
<field name="name" primary-key="true"/>
<field name="emps">
<collection element-type="Employee">
<extension vendor-name="sunw" key="element-inverse" value="dept"/>
</collection>
</field>
</class>
</package>
</jdo>
```

19 Portability Guidelines

One of the objectives of JDO is to allow an application to be portable across multiple JDO implementations. This Chapter summarizes portability rules that are expressed elsewhere in this document. If all of these programming rules are followed, then the application will work in any JDO compliant implementation.

19.1 Optional Features

These features may be used by the application if the JDO vendor supports them. Since they are not required features, a portable application must not use them.

19.1.1 Optimistic Transactions

Optimistic transactions are enabled by the `PersistenceManagerFactory` or `Transaction` method `setOptimistic(true)`. JDO implementations that do not support optimistic transactions throw `JDOUnsupportedOptionException`.

19.1.2 Nontransactional Read

Nontransactional read is enabled by the `PersistenceManagerFactory` or `Transaction` method `setNontransactionalRead(true)`. JDO implementations that do not support nontransactional read throw `JDOUnsupportedOptionException`.

19.1.3 Nontransactional Write

Nontransactional write is enabled by the `PersistenceManagerFactory` or `Transaction` method `setNontransactionalWrite(true)`. JDO implementations that do not support nontransactional write throw `JDOUnsupportedOptionException`.

19.1.4 Transient Transactional

Transient transactional instances are created by the `PersistenceManager` `makeTransactional(Object)`. JDO implementations that do not support transient transactional throw `JDOUnsupportedOptionException`.

19.1.5 RetainValues

A portable application should run the same regardless of the setting of the `retainValues` flag.

19.1.6 IgnoreCache

A portable application should set this flag to `false`. The results of iterating `Extents` and executing queries might be different among different implementations.

19.2 Object Model

References among `PersistenceCapable` classes must be defined as First Class Objects in the model.

SCO instances must not be shared among multiple persistent instances.

Arrays must not be shared among multiple persistent instances.

If arrays are passed by reference outside the defining class, the owning persistent instance must be notified via `jdoMakeDirty`.

The application must not depend on any sharing semantics of immutable class objects.

The application must not depend on knowing the exact class of an SCO instance, as they may be substituted by a subclass of the type.

Persistence-capable classes must not contain final non-static fields or methods or fields that start with "jdo".

19.3 JDO Identity

Applications must be aware that support for application identity and datastore identity are optional, and some implementations might support only one of these identity types. The supported identity type(s) of the implementation should be checked by using the `supportedOptions` method of `PersistenceManagerFactory`.

Applications must construct only `ObjectId` instances for classes that use application-defined JDO identity, or use the `PersistenceManager.getObjectIdClass` to obtain the `ObjectId` class.

Classes that use application identity must only use key field types of primitive, `String`, `Date`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigDecimal`, or `BigInteger`.

Applications must only compare `ObjectId` instances from different JDO implementations for classes that use application-defined JDO identity.

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

Key fields can be defined only in the least-derived persistence-capable class in an inheritance hierarchy. All of the classes in the hierarchy use the same key class.

A JDO implementation might not support changing primary key field values (which has the effect of changing the primary key of the underlying datastore instance). Portable applications do not change primary key fields.

19.4 PersistenceManager

To be portable, instances of `PersistenceManager` must be obtained from a `PersistenceManagerFactory`, and not by construction. The recommended way to instantiate a `PersistenceManagerFactory` is to use the `JDOHelper.getPersistenceManagerFactory(Properties)` method.

19.5 Query

Using a query language other than JDOQL is not portable.

A query must constrain all variables used in any expressions with a `contains` clause referencing a persistent field of a `PersistenceCapable` class.

Not all datastores allow storing null-valued collections. Portable queries on these collections should use `isEmpty()` instead of comparing to `null`.

Portable queries must not use non-persistent, final, or static field names in filter expressions.

Portable queries must pass persistent or transactional instances as parameters of persistence-capable types.

19.6 XML metadata

Portable applications will define all persistence-capable classes in the XML metadata.

19.7 Life cycle

Portable applications will not depend on requiring instances to be hollow or persistent-nontransactional, or to remain non-transactional in a transaction.

19.8 JDOHelper

Portable applications will use JDOHelper for state interrogations of instances of persistence-capable classes and for determining if an instance is of a persistence-capable class.

19.9 Transaction

Portable applications must not depend on isolation levels stronger than read-committed provided by the underlying datastore. Some fields might be read at different times by the JDO implementation, and there is no guarantee as to read consistency compared to previously read data. A JDO persistence-capable instance might contain fields instantiated by multiple datastore accesses, with no guarantees of consistency (read-committed isolation level).

Readers primarily interested in developing applications with the JDO API can ignore the following chapters. Skip to 22 – JDOPermission.

20 JDO Reference Enhancer

This chapter specifies the JDO Reference Enhancement, which specifies the contract between JDO persistence-capable classes and JDO `StateManager` in the runtime environment. The JDO Reference Enhancer modifies persistence-capable classes to run in the JDO environment and implement the required contract. The resulting classes, hereinafter referred to as enhanced classes, implement a contract used by the `JDOHelper`, the `JDOImplHelper`, and the `StateManager` classes.

The JDO Reference Enhancer is just one possible implementation of the JDO Reference Enhancement contract. Tools may instead preprocess or generate source code to create classes that implement this contract.

NOTE: This chapter is not intended to be used by application programmers. It is for use only by implementations. Applications should use the methods defined in class `JDOHelper` instead of these methods and fields.

20.1 Overview

The JDO Reference Enhancer will be used to modify each persistence-capable class before using that persistence-capable class with the `ReferenceImplementationPersistenceManager` in the Java VM. It might be used before class loading or during the class loading process.

The JDO Reference Enhancer transforms the class by making specific changes to the class definition to enable the state of any persistent instances to be synchronized with the representation of the data in the datastore.

Tools that generate source code or modify the Java source code files must generate classes that meet the defined contract in this chapter.

The Reference Enhancer provides an implementation for the `PersistenceCapable` interface.

20.2 Goals

The following are the goals for the JDO Reference Enhancer:

- Binary compatibility and portability of application classes among JDO vendor implementations
- Binary compatibility between application classes enhanced by different JDO vendors at different times.
- Minimal intrusion into the operation of the class and class instances
- Provide metadata at runtime without requiring implementations to be granted `reflect` permission for non-private fields

- Values of fields can be read and written directly without wrapping code with accessors or mutators (`field1 += 13` is allowed, instead of requiring the user to code `setField1(getField1() + 13)`)
- Navigation from one instance to another uses natural Java syntax without any requirement for explicit fetching of referenced instances
- Automatically track modification of persistent instances without any explicit action by the application or component developer
- Highest performance for transient instances of persistence-capable classes
- Support for all class and field modifiers
- Transparent operation of persistent and transient instances as seen by application components and persistence-capable classes
- Shared use of persistence-capable classes (utility components) among multiple JDO `PersistenceManager` instances in the same Java VM
- Preservation of the security of instances of `PersistenceCapable` classes from unauthorized access
- Support for debugging enhanced classes by line number

20.3 Enhancement: Architecture

The reference enhancement of `PersistenceCapable` classes has the primary objective of preserving transparency for the classes. Specifically, accesses to fields in the JDO instance are mediated to allow for initializing values of fields from the associated values in the datastore and for storing the values of fields in the JDO instance into the associated values in the datastore at transaction boundaries.

To avoid conflicts in the name space of the `PersistenceCapable` classes, all methods and fields added to the `PersistenceCapable` classes have the “jdo” prefix.

Enhancement might be performed at any time prior to use of the class by the application. During enhancement, special JDO class metadata must be available if any non-default actions are to be taken. The metadata is in XML format .

Specifically, the following will require access to special class metadata at class enhancement time, because these are not the defaults:

- classes are to use primary key or non-managed object identity;
- fields declared as transient in the class definition are to be persistent in the datastore;
- fields not declared as transient in the class definition are to be non-persistent in the datastore;
- fields are to be transactional non-persistent;
- fields with domains of references to `PersistenceCapable` classes are to be part of the default fetch group;
- fields with domains of primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) or primitive wrapper types (`Boolean`, `Char`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) are not to be part of the default fetch group;
- fields with domains of `String` are not to be part of the default fetch group;

- fields with domains of array types are to be part of the default fetch group.

Enhancement makes changes to two categories of classes: persistence-capable and persistence-aware. Persistence-capable classes are those whose instances are allowed to be stored in a JDO-managed datastore. Persistence aware classes are those that while not necessarily persistence-capable themselves, contain references to managed fields of classes that are persistence-capable. Thus, persistence-capable classes may also be persistence-aware.

To preserve the security of instances of `PersistenceCapable` classes, access restrictions to fields before enhancement will be propagated to accessor methods after enhancement. Further, to become the delegate of field access (`StateManager`) the caller must be authorized for `JDOPermission`.

A JDO implementation must interoperate with classes enhanced by the Reference Enhancer and with classes enhanced with other Vendor Enhancers. Additionally, classes enhanced by any Vendor Enhancers must interoperate with the Reference Implementation.

Name scope issues are minimized because the Reference Enhancement contract adds methods and fields that begin with “jdo”, while methods and fields added by Vendor Enhancers must not begin with “jdo”. Instead, they may begin with “sunwjdo”, “exlnjdo” or other string that includes a vendor-identifying name and the “jdo” string.

Debugging by source line number must be preserved by the enhancement process. If any code modification within a method body changes the byte code offsets within the method, then the line number references of the method must be updated to reflect the change.

The Reference Enhancer makes the following changes to the least-derived (topmost) persistence-capable classes:

- adds a field named `jdoStateManager`, of type `javax.jdo.spi.StateManager` to associate each instance with zero or one instance of JDO `StateManager`;
- adds a synchronized method `jdoReplaceStateManager` (to replace the value of the `jdoStateManager`), which invokes security checking for declared `JDOPermission`;
- adds a field named `jdoFlags` of type `byte` in the least-derived persistence capable class, to distinguish readable and writable instances from non-readable and non-writable instances;
- adds a method `jdoReplaceFlags` to require the instance to request an updated value for the `jdoFlags` field from the `StateManager`;
- adds methods to implement status query methods by delegating to the `StateManager`;
- adds method `jdoReplaceFields(int[])` to obtain values of specified fields from the `StateManager` and cache the values in the instance;
- adds method `jdoProvideFields(int[])` to supply values of specific fields to the `StateManager`;
- adds a method `void jdoCopyFields(Object other, int[] fieldNumbers)` to allow the `StateManager` to manage multiple images of the persistence capable instance;
- adds a method `jdoPreSerialize` to load all non-transient fields into the instance prior to serialization;

The Reference Enhancer makes the following changes to least-derived (topmost) persistence-capable classes and classes that declare an `objectId-class` in their xml:

- adds methods `jdoCopyKeyFieldsToObjectId(PersistenceCapable pc, Object oid)` and `jdoCopyKeyFieldsToObjectId(ObjectIdFieldSupplier fs, Object oid)`.
- adds methods `jdoCopyKeyFieldsFromObjectId(Object oid)` and `jdoCopyKeyFieldsFromObjectId(ObjectIdFieldConsumer fc, Object oid)`.
- adds a method `jdoNewObjectIdInstance()` which creates an instance of the `jdo ObjectId` for this class.

The Reference Enhancer makes the following changes to all classes:

- adds “implements `javax.jdo.spi.PersistenceCapable`” to the class definition;
- adds two methods `jdoNewInstance`, one of which takes a parameter of type `StateManager`, to be used by the implementation when a new persistent instance is required (this method allows a performance optimization), and the other takes a parameter of type `StateManager` and a parameter of an `ObjectId` for key field initialization;
- adds method `jdoReplaceField(int)` to obtain values of specified fields from the `StateManager` and cache the values in the instance;
- adds method `jdoProvideField(int)` to supply values of specific fields to the `StateManager`;
- adds an accessor method and mutator method for each field declared in the class, which delegates to the `StateManager` for values;
- leaves the modifiers of all persistent fields the same as the unenhanced class to allow the enhanced classes to be used for compilation of other classes;
- adds a method `jdoCopyField(<class> other, int fieldNumber)` to allow the `StateManager` to manage multiple images of the persistence capable instance;
- adds a method `jdoGetManagedFieldCount()` to manage the numbering of fields with respect to inherited managed fields.
- adds a field `jdoInheritedFieldCount`, which is set at class initialization time to the returned value of `super.jdoGetManagedFieldCount()`.
- adds fields `jdoFieldNames`, `jdoFieldTypes`, and `jdoFieldFlags`, which contain the names, types, and flags of managed fields.
- adds field `Class jdoPersistenceCapableSuperclass`, which contains the `Class` of the `PersistenceCapable` superclass.
- adds a static initializer to register the class with the `JDOImplHelper`.
- adds a field `serialVersionUID` if it does not already exist, and calculates its initial value based on the non-enhanced class definition.

Enhancement makes the following changes to persistence aware classes:

- modifies executable code that accesses fields of `PersistenceCapable` classes not known to be not managed, replacing `getField` and `putField` calls with calls to the generated accessor and mutator methods.

20.4 Inheritance

Enhancement allows a class to manage the persistent state only of declared fields. It is a future objective to allow a class to manage fields of a non-persistence capable superclass.

Fields that hide inherited fields (because they have the same name) are fully supported. The enhancer delegates accesses of inherited hidden fields to the appropriate class by referencing the appropriate method implemented in the declaring class.

All persistence capable classes in the inheritance hierarchy must use the same kind of JDO identity.

20.5 Field Numbering

Enhancement assigns field numbers to all managed (transactional or persistent) fields. Generated methods and fields that refer to fields (`jdoFieldNames`, `jdoFieldTypes`, `jdoFieldFlags`, `jdoGetManagedFieldCount`, `jdoCopyFields`, `jdoMakeDirty`, `jdoProvideField`, `jdoProvideFields`, `jdoReplaceField`, and `jdoReplaceFields`) are generated to include both transactional and persistent fields.

Relative field numbers are calculated at enhancement time. For each persistence capable class the enhancer determines the declared managed fields. To calculate the relative field number, the declared fields array is sorted by field name. Each managed field is assigned a relative field number, starting with zero.

Absolute field numbers are calculated at runtime, based on the number of inherited managed fields, and the relative field number. The absolute field number used in method calls is the relative field number plus the number of inherited managed fields.

The absolute field number is used in method calls between the `StateManager` and `PersistenceCapable`; and in the reference implementation, between the `StateManager` and `StoreManager`.

20.6 Serialization

Serialization of a transient instance results in writing an object graph of objects connected via non-transient fields. The explicit intent of JDO enhancement of serializable classes is to permit serialization of transient instances or persistent instances to a format that can be deserialized by either an enhanced or non-enhanced class.

When the `writeObject` method is called on a class to serialize it, all fields not declared as transient must be loaded into the instance. This function is performed by the enhancer-generated method `jdoPreSerialize`. This method simply delegates to the `StateManager` to ensure that all persistent non-transient fields are loaded into the instance. [Fields not declared as transient and not declared as persistent must have been loaded by the `PersistenceCapable` class an application-specific way.]

The `jdoPreSerialize` method need be called only once for a persistent instance. Therefore, the `writeObject` method in the least-derived `pc` class that implements `Serializable` in the inheritance hierarchy needs to be modified or generated to call it.

If a standard serialization is done to an enhanced class instance, the fields added by the enhancer will not be serialized because they are declared to be `transient`.

To allow a non-enhanced class to deserialize the stream, the `serialVersionUID` for the enhanced and non-enhanced classes must be identical. If the `serialVersionUID` field does not already exist in the non-enhanced class, the enhancer will calculate it (excluding any enhancer-generated fields or methods) and add it to the enhanced class.

If a `PersistenceCapable` class is assignable to `java.io.Serializable` but its persistence-capable superclass is not, then the enhancer will modify the class in the following way:

- if the class does not contain implementations of `writeObject`, or `writeReplace`, then the enhancer will generate `writeObject`. Fields that are required to be present during serialization operations will be explicitly instantiated by the generated method `jdoPreSerialize`, which will be called by the enhancer-generated `writeObject`.
- if the class contains an implementation of `writeObject` or `writeReplace`, it will be changed to call `jdoPreSerialize` prior to any user-written code in the method.

If a `PersistenceCapable` class is assignable to `java.io.Serializable`, then the non-transient fields might be instantiated prior to serialization. However, the closure of instances reachable from this instance might include a large part of instances in the data-store.

The results of restoring a serialized persistent instance graph is a graph of interconnected transient instances. The method `readObject` is not enhanced, as it deals only with transient instances.

20.7 Cloning

If a standard clone is made of a persistent instance, the `jdoFlags` and `jdoStateManager` fields will also be cloned. The clone will eventually invoke the `StateManager` if the source of the cloned instance is not transient. This condition will be detected by the runtime, but disconnecting the clone is a convoluted process. To avoid this situation where possible, the enhancer modifies the cloning behavior by modifying certain methods that invoke `clone`, setting these two fields to indicate that the clone is a transient instance. Otherwise, all of the fields in the clone contain the standard shallow copy of the fields of the cloned instance.

The reference enhancement will modify the `clone()` method in the persistence-capable root class (the least-derived (topmost) `PersistenceCapable` class) to reset these two fields immediately after returning from `super.clone()`. This caters for the normal case where `clone` methods in subclasses call `super.clone()` and the clone is disconnected immediately after being cloned.

This technique does not address these cases:

- A non-persistence-capable superclass `clone` method calls a runtime method (for example, `makePersistent`) on the newly created clone. In this case, the `makePersistent` will succeed, but the `clone` method in the persistence-capable subclass will disconnect the clone, thereby undoing the `makePersistent`. Thus, calling any life cycle change methods with the clone as an argument is not permitted in `clone` methods.

- Where there is no clone method declared in the persistence-capable root class, the clone will not be disconnected, and the runtime will disconnect the clone the first time the `StateManager` is called by the clone.

20.8 Introspection (Java core reflection)

No changes are made to the behavior of introspection. The current state of all fields is exposed to the reflection APIs.

This is not at all what some users might expect. It is a future objective to more gracefully support introspection of fields in persistent instances of persistence capable classes.

20.9 Field Modifiers

Fields in `PersistenceCapable` classes are treated by the enhancer in one of several ways, based on their modifiers as declared in the Java language and their enhanced modifiers as declared by the `PersistenceCapable` `MetaData`.

These modifiers are orthogonal to the modifiers defined by the Java language. They have default values based on modifiers defined in the class for the fields. They may be specified in the XML metadata used at enhancement time.

20.9.1 Non-persistent

Non-persistent fields are ignored by the enhancer. They are assumed to lie outside the domain of persistence. They might be changed at will by any method based only on the private/protected/public modifiers. There is no enhancement of accesses to non-persistent fields.

The default modifier is non-persistent for fields identified as transient in the class declaration.

20.9.2 Transactional non-persistent

Transactional non-persistent fields are non-persistent fields whose values are saved and restored during rollback. Their values are not stored in the datastore. There is no enhancement of read accesses to transactional non-persistent fields. Write accesses are always mediated (the `StateManager` is called on write).

20.9.3 Persistent

Persistent fields are fields whose values are synchronized with values in the datastore. The synchronization is performed transparent to the methods in the `PersistenceCapable` class.

The default persistence-modifier for fields is based on their modifiers and type, as detailed in the XML metadata chapter.

The modification to the class by the enhancer depends on whether the persistent field is a member of the default fetch group.

If the persistent field is a member of the default fetch group, then the enhanced code behaves as follows. The constant values `READ_OK`, `READ_WRITE_OK`, and `LOAD_REQUIRED` are defined in interface `PersistenceCapable`.

- for read access, `jdoFlags` is checked for `READ_OK` or `READ_WRITE_OK`. If it is then the value in the field is retrieved. If it is not, then the `StateManager` instance is requested to load the value of the field from the datastore, which might cause the `StateManager` to populate values of all default fetch group fields in

the instance, and other values as defined by the JDO vendor policy. This behavior is not required, but optional. If the `StateManager` chooses, it may simply populate the value of the specific field requested. Upon conclusion of this process, the `jdoFlags` value might be set by the `StateManager` to `READ_OK` and the value of the field is retrieved. If not all fields in the default fetch group were populated, the `StateManager` must not set the `jdoFlags` to be `READ_OK`.

- for write access, `jdoFlags` is checked for `READ_WRITE_OK`. If it is `READ_WRITE_OK`, then the value is stored in the field. If it is not `READ_WRITE_OK`, then the `StateManager` instance is requested to load the state of the values from the datastore, which might cause the `StateManager` to populate values of all default fetch group fields in the instance. Upon conclusion of the load process, the `jdoFlags` value might be set by the `StateManager` to `READ_WRITE_OK` and the value of the field is stored.

If the persistent field is not a member of the default fetch group, then each read and write access to the field is delegated to the `StateManager`. For read, the value of the field is obtained from the `StateManager`, stored in the field, and returned to the caller. For write, the proposed value is given to the `StateManager`, and the returned value from the `StateManager` is stored in the field.

The enhanced code that fetches or modifies a field that is not in the default fetch group first checks to see if there is an associated `StateManager` instance and if not (the case for transient instances) the access is allowed without intervention.

20.9.4 PrimaryKey

Primary key fields are not part of the default fetch group; all changes to the field can be intercepted by the `StateManager`. This allows special treatment by the implementation if any primary key fields are changed by the application.

Primary key fields are always available in the instance, regardless of the state. Therefore, read access to these fields is never mediated.

20.9.5 Embedded

Fields identified as embedded in the XML metadata are treated as containing embedded instances. The default for `Array`, `Collection`, and `Map` types is embedded. This is to allow JDO implementations to map `PersistenceCapable` field types to embedded objects (aggregation by containment pattern).

20.9.6 Null-value

Fields of `Object` types might be mapped to datastore elements that do not allow null values. The default behavior “none” is that no special treatment is done for null-valued fields. In this case, null-valued fields throw a `JDOUserException` when the instance is flushed to the datastore and the datastore does not support null values.

However, the treatment of null-valued fields can be modified by specifying the behavior in the XML metadata. The null-value setting of “default” is used when the default value for the datastore element is to be used for null-valued fields.

If the application requires non-null values to be stored in this field, then the setting should be “exception”, which throws a `JDOUserException` if the value of the field is null at the time the instance is stored in the datastore.

For example, if a field of type `Integer` is mapped to a datastore `int` value, committing an instance with a field value of null where the null-value setting is “default” will result in

a zero written to the datastore element. Similarly, a null-valued `String` field would be written to the datastore as an empty (zero length) `String` where the null-value setting is “default”.

20.10 Treatment of standard Java field modifiers

20.10.1 Static

Static fields are ignored by the enhancer. They are not initialized by JDO; accesses to values are not mediated.

20.10.2 Final

Final fields are treated as non-persistent and non-transactional by the enhancer. Final fields are initialized only by the constructor, and their values cannot be changed after construction of the instance. Therefore, their values cannot be loaded or stored by JDO; accesses are not mediated.

This treatment might not be what users expect; therefore, final fields are not supported as persistent or transactional instance fields, final static fields are supported by ignoring them.

20.10.3 Private

Private fields are accessed only by methods in the class itself. JDO handles private fields according to the semantic that values are stored in private fields by the enhancement-generated `jdoSetXXX` methods or `jdoReplaceField`, which become part of the class definition. The enhancement-generated `jdoGetXXX` or `jdoProvideField` methods, which become part of the class definition, load values from private fields.

20.10.4 Public, Protected

Public fields are not recommended to be persistent in persistence capable classes. Classes that make reference to persistent public fields (persistence aware) must be enhanced themselves prior to execution. Protected fields and fields without an explicit access modifier (commonly referred to as package access) may be persistent.

Users must enhance all classes, regardless of package, that reference any persistent or transactional field.

20.11 Fetch Groups

Fetch groups represent a grouping of fields that are retrieved from the datastore together. Typically, a datastore associates a number of data values together and efficiently retrieves these values. Other values require extra method calls to retrieve.

For example, in a relational database, the `Employee` table defines columns for `Employee id`, `Name`, and `Position`. These columns are efficiently retrieved with one data transfer request. The corresponding fields in the `Employee` class might be part of the default fetch group.

Continuing this example, there is a column for `Department dept`, defined as a foreign key from the `Employee` table to the `Department` table, which corresponds to a field in the `Employee` class named `dept` of type `Department`. The runtime behavior of this field depends on the mapping to the `Department` table. The reference might be to a derived class and it might be expensive to determine the class of the `Department` instance. Therefore, the `dept` field will not be defined as part of the default fetch group, even though the foreign key that

implements the relationship might be fetched when the Employee is fetched. Rather, the value for the dept field will be retrieved from the `StateManager` every time it is requested. Similarly, the `StateManager` will be called for each modification of the value of dept. The `jdoFlags` field is the indicator of the state of the default fetch group.

20.12 jdoFlags Definition

The value of the `jdoFlags` field is entirely determined by the `StateManager`. The `StateManager` calls the `jdoReplaceFlags` method to inform the persistence capable class to retrieve a new value for the `jdoFlags` field. The values permitted are constants defined in the interface `PersistenceCapable`: `READ_OK`, `READ_WRITE_OK`, and `LOAD_REQUIRED`.

During the transition from transient to a managed life cycle state, the `jdoFlags` field is set to `LOAD_REQUIRED` by the persistence capable instance, to indicate that the instance is not ready. During the transition from a managed state to transient, the `jdoFlags` field is set to `READ_WRITE_OK` by the persistence capable instance, to indicate that the instance is available for read and write of any field.

The `jdoFlags` field is a byte with three possible values and associated meanings:

- 0 - `READ_WRITE_OK`: the values in the default fetch group can be read or written without intermediation of the associated `StateManager` instance.
- -1 - `READ_OK`: the values in the default fetch group can be read but not written without intermediation of the associated `StateManager` instance.
- 1 - `LOAD_REQUIRED`: the values in the default fetch group cannot be accessed, either for read or write, without intermediation of the associated `StateManager` instance.

20.13 Exceptions

Generated methods validate the state of the persistence-capable class and the arguments to the method.

If an argument is illegal, then `IllegalArgumentException` is thrown. For example, an illegal field number argument is less than zero or greater than the number of managed fields.

Some methods require a non-null state manager. In these cases, if the `jdoStateManager` is null, then `IllegalStateException` is thrown.

20.14 Modified field access

The enhancer modifies field accesses to guarantee that the values of fields are retrieved from the datastore prior to application usage.

For any field access that reads the value of a field, the `getField` byte code is replaced with a call to a generated local method, `jdoGetXXX`, which determines based on the kind of field (default fetch group or not) and the state of the `jdoFlags` whether to call the `StateManager` with the field number needed.

For any field access that stores the new value of a field, the `putField` byte code is replaced with a call to a generated local method, `jdoSetXXX`, which determines based on the kind

of field (default fetch group or not) and the state of the `jdoFlags` whether to call the `StateManager` with the field number needed. A JDO implementation might perform field validation during this operation and might throw a `JDOUserException` if the value of the field does not meet the criterion.

The following table specifies the values of the `jdoFieldFlags` for each type of mediated

Table 5: Field access mediation

field type	read access	write access	flags
transient transactional	not checked	checked	CHECK_WRITE
primary key	not checked	mediated	MEDIATE_WRITE
default fetch group	checked	checked	CHECK_READ + CHECK_WRITE
non-default fetch group	mediated	mediated	MEDIATE_READ + MEDIATE_WRITE

field.

not checked: access is always granted

checked: the condition of `jdoFlags` is checked to see if access should be mediated

mediated: access is always mediated (delegated to the `StateManager`)

flags: the value in the `jdoFieldFlags` field

The flags are defined in `PersistenceCapable` and may be combined only as in the above table (`SERIALIZABLE` may be combined with any other flags):

- 1 - CHECK_READ
- 2 - MEDiate_READ
- 4 - CHECK_WRITE
- 8 - MEDiate_WRITE
- 16 - SERIALIZABLE

20.15 Generated fields in least-derived `PersistenceCapable` class

These fields are generated only in the least-derived (topmost) class in the inheritance hierarchy of persistence-capable classes.

```
protected transient javax.jdo.spi.StateManager jdoStateManager;
This field contains the managing StateManager instance, if this instance is being managed.
protected transient byte jdoFlags;
```

20.16 Generated fields in all `PersistenceCapable` classes

The following fields are generated in all persistence-capable classes.

```
private final static int jdoInheritedFieldCount;
```

This field is initialized at class load time to be the number of fields managed by the super-classes of this class, or to zero if there is no persistence capable superclass.

```
private final static String[] jdoFieldNames;
```

This field is initialized at class load time to an array of names of persistent and transactional fields. The position in the array is the relative field number of the field.

```
private final static Class[] jdoFieldTypes;
```

This field is initialized at class load time to an array of types of persistent and transactional fields. The position in the array is the relative field number of the field.

```
private final static byte[] jdoFieldFlags;
```

This field is initialized at class load time to an array of flags indicating the characteristics of each persistent and transactional field.

```
private final static Class jdoPersistenceCapableSuperclass;
```

This field is initialized at class load time to the class instance of the PersistenceCapable superclass, or null if there is none.

```
private final static long serialVersionUID;
```

This field is declared only if it does not already exist, and it is initialized to the value that would obtain prior to enhancement.

Generated static initializer

The generated static initializer uses the values for `jdoFieldNames`, `jdoFieldTypes`, `jdoFieldFlags`, and `jdoPersistenceCapableSuperclass`, and calls the static `registerClass` method in `JDOImplHelper` to register itself with the runtime environment. If the class is abstract, then it does not register a helper instance. If the class is not abstract, it registers a newly constructed instance.

The generated static initialization code is placed after any user-defined static initialization code.

20.17 Generated methods in least-derived PersistenceCapable class

These methods are declared in interface `PersistenceCapable`.

```
public final boolean jdoIsPersistent();
```

```
public final boolean jdoIsTransactional();
```

```
public final boolean jdoIsNew();
```

```
public final boolean jdoIsDirty();
```

```
public final boolean jdoIsDeleted();
```

These methods check if the `jdoStateManager` field is null. If so, they return false. If not, they delegate to the corresponding method in `StateManager`.

```
public final void jdoMakeDirty (String fieldName);
```

This method checks if the `jdoStateManager` field is null. If so, it returns silently. If not, it delegates to the `makeDirty` method in `StateManager`.

```
public final PersistenceManager jdoGetPersistenceManager();
```

This method checks if the `jdoStateManager` field is null. If so, it returns null. If not, it delegates to the `getPersistenceManager` method in `StateManager`.

```
public final Object jdoGetObjectId();
```

```
public final Object jdoGetTransactionalObjectId();
```

These methods check if the `jdoStateManager` field is null. If so, they return null. If not, they delegate to the corresponding method in `StateManager`.

```
public synchronized final void jdoReplaceStateManager (StateM-
anager sm);
```

NOTE: This method will be called by the `StateManager` on state changes when transitioning an instance from transient to a managed state, and from a managed state to transient.

This method is implemented as synchronized to resolve race conditions, if more than one `StateManager` attempts to acquire ownership of the same `PersistenceCapable` instance.

If the current `jdoStateManager` is not null, this method replaces the current value for `jdoStateManager` with the result of calling `jdoStateManager.replacingStateManager(this, sm)`. If successful, the method ends. If the change was not requested by the `StateManager`, then the `StateManager` throws a `JDOUserException`.

If the current `jdoStateManager` field is null, then a security check is performed by calling `JDOImplHelper.checkAuthorizedStateManager` with the `StateManager` parameter `sm` passed as the parameter to the check. Thus, only `StateManager` instances in code bases authorized for `JDOPermission("setStateManager")` are allowed to set the `StateManager`. If the security check succeeds, the `jdoStateManager` field is set to the value of the parameter `sm`, and the `jdoFlags` field is set to `LOAD_REQUIRED` to indicate that mediation is required.

```
public final void jdoReplaceFlags ();
```

NOTE: This method will be called by the `StateManager` on state changes when transitioning an instance from a managed state to transient.

If the current `jdoStateManager` field is null, then this method silently returns with no effect.

If the current `jdoStateManager` is not null, this method replaces the current value for `jdoFlags` with the result of calling `jdoStateManager.replacingFlags(this)`.

```
public final void jdoReplaceFields (int[] fields);
```

For each field number in the `fields` parameter, `jdoReplaceField` method is called.

```
public final void jdoProvideFields (int[] fields);
```

For each field number in the `fields` parameter, `jdoProvideField` method is called.

```
protected final void jdoPreSerialize();
```

This method is called by the generated or modified `writeObject` to allow the instance to fully populate serializable fields. This method delegates to the `StateManager` method `preSerialize` so that fields can be fetched by the JDO implementation prior to serialization. If the `jdoStateManager` field is null, this method returns with no effect.

20.18 Generated methods in `PersistenceCapable` root classes and all classes that declare `objectid-class` in xml metadata:

```
public void jdoCopyKeyFieldsToObjectId (ObjectIdFieldSupplier
fs, Object oid)
```

This method is called by the JDO implementation (or implementation helper) to populate key fields in object id instances. If this class is not the `PersistenceCapable` root class, it first calls the method of the same name in the root class. Then, for each key field declared in the metadata, this method calls the object id field supplier and stores the result in the oid instance.

If the oid parameter is not assignment compatible with the object id class of this instance, then `ClassCastException` is thrown. If this class does not use application identity, then this method silently returns.

```
public void jdoCopyKeyFieldsToObjectId (Object oid)
```

This method is called by the JDO implementation (or implementation helper) to populate key fields in object id instances from persistence-capable instances. This might be used to implement `getObjectId` or `getTransactionalObjectId`. If this class is not the `PersistenceCapable` root class, it first calls the method of the same name in the root class. Then, for each key field declared in the metadata, this method copies the value of the key field to the oid instance.

If the oid parameter is not assignment compatible with the object id class of this instance, then `ClassCastException` is thrown. If this class does not use application identity, then this method silently returns.

```
public void jdoCopyKeyFieldsFromObjectId(ObjectIdFieldConsumer fc, Object oid)
```

This method is called by the JDO implementation (or implementation helper) to export key fields from object id instances. If this class is not the `PersistenceCapable` root class, it first calls the method of the same name in the root class. Then, for each key field declared in the metadata, this method passes the value of the key field in the oid instance to the store method of the object id field consumer.

If the oid parameter is not assignment compatible with the object id class of this instance, then `ClassCastException` is thrown. If this class does not use application identity, then this method silently returns.

```
protected void jdoCopyKeyFieldsFromObjectId (Object oid)
```

This method is called by the `jdoNewInstance(Object oid)` method. If this class is not the `PersistenceCapable` root class, it first calls the method of the same name in the root class. Then, for each key field declared in the metadata, this method copies the value of the key field in the oid instance to the key field in this instance.

If the oid parameter is not assignment compatible with the object id class of this instance, then `ClassCastException` is thrown. If this class does not use application identity, then this method silently returns.

```
public Object jdoNewObjectIdInstance();
```

```
public Object jdoNewObjectIdInstance(String str);
```

NOTE: This method is called by the JDO implementation (or implementation helper) to populate key fields in object id instances.

If this class uses application identity, then this method returns a new instance of the `ObjectId` class. Otherwise, `null` is returned.

20.19 Generated methods in all `PersistenceCapable` classes

```
public PersistenceCapable jdoNewInstance(StateManager sm);
```

This method uses the default constructor, assigns the `sm` parameter to the `jdoStateManager` field, and assigns `LOAD_REQUIRED` to the `jdoFlags` field. If the class is abstract, a `JDOFatalInternalException` is thrown.

```
public PersistenceCapable jdoNewInstance(StateManager sm, Object objectid);
```

This method uses the default constructor, assigns the `StateManager` parameter to the `jdoStateManager` field, assigns `LOAD_REQUIRED` to the `jdoFlags` field, and copies the key fields from the `objectid` parameter. If the class is abstract, a `JDOFatalInternalException` is thrown. If the `objectid` parameter is not of the correct class, then `ClassCastException` is thrown.

```
protected static int jdoGetManagedFieldCount();
```

This method returns the number of managed fields declared by this class plus the number inherited from all superclasses. This method is generated in the class to allow the class to determine at runtime the number of inherited fields, without having introspection code in the enhanced class.

```
final static mmm ttt jdoGet<field>(<class> instance);
```

The generated `jdoGet` methods have exactly the same stack signature as the byte code `getField`. They return the value of one specific field. The field returned was either cached in the instance or retrieved from the `StateManager`.

The name of the generated method is constructed from the field name. This allows for hidden fields to be supported explicitly, and for classes to be enhanced independently.

The modifier `mmm` is the same access modifier as the corresponding field in the unenhanced class. The return type `ttt` is the same type as the corresponding field in the unenhanced class.

The generated code depends on the type of field:

- If the field is `CHECK_READ`, then the method first checks to see if the `jdoFlags` field is not `LOAD_REQUIRED`. If so, the value of the field is returned. If not, then the value of `jdoStateManager` is checked. If it is `null`, the value of the field is returned. If non-`null`, then method `isLoaded` is called on the `jdoStateManager`. If the result of `isLoaded` is `true`, then the value of the field is returned. If the result of `isLoaded` is `false`, then the result of method `getXXXField` on the `jdoStateManager` is returned.
- If the field is `MEDIATE_READ`, then the value of `jdoStateManager` is checked. If it is `null`, the value of the field is returned. If non-`null`, then method `isLoaded` is called on the `jdoStateManager`. If the result of `isLoaded` is `true`, then the value of the field is returned. If the result of `isLoaded` is `false`, then the result of method `getXXXField` on the `jdoStateManager` is returned.
- If the field is neither of the above, then the value of the field is returned.

```
final static mmm void jdoSet<field> (<class> instance, ttt newValue);
```

The generated `jdoSet` methods have exactly the same stack signature as the byte code `putField`. They set the value of one specific field. The field might be provided to the `StateManager`.

The name of the generated method is constructed from the field name. This allows for hidden fields to be supported explicitly, and for classes to be enhanced independently.

The modifier `mmm` is the same access modifier as the corresponding field in the unenhanced class. The type `ttt` is the same type as the corresponding field in the unenhanced class.

The generated code depends on the type of field:

- If the field is `CHECK_WRITE`, then the method first checks to see if the `jdoFlags` field is `READ_WRITE_OK`. If so, then the field is set to the new value. If not, then the value of `jdoStateManager` is checked. If it is `null`, the value of the field is set to the new value. If non-`null`, then method `setXXXField` is executed on the `jdoStateManager`, passing the new value.
- If the field is `MEDIATE_WRITE`, then the value of `jdoStateManager` is checked. If it is `null`, then the field is set to the parameter. If non-`null`, then method `setXXXField` is executed on the `jdoStateManager`, passing the new value.
- If the field is neither of the above, then the value of the field is set to the new value.

```
public void jdoReplaceField (int field);
```

NOTE: This method is used by the `StateManager` to store values from the datastore into the instance. If there is no `StateManager` (the `jdoStateManager` field is `null`), then this method throws `JDOFatalInternalException`.

This method calls the `StateManager` `replacingXXXField` to get a new value for one field from the `StateManager`.

The field number is examined to see if it is a declared field or an inherited field. If it is inherited, then the call is delegated to the superclass. If it is declared, then the appropriate `StateManager` `replacingXXXField` method is called, which retrieves the new value for the field.

If the field is out of range (less than zero or greater than the number of managed fields in the class) then a `JDOFatalInternalException` is thrown.

```
public void jdoProvideField (int field);
```

NOTE: This method is used by the `StateManager` to retrieve values from the instance, during flush to the datastore or for in-memory query processing. If there is no `StateManager` (the `jdoStateManager` field is `null`), then this method throws `JDOFatalInternalException`.

This method calls the `StateManager` `providedXXXField` method to supply the value of the specified field to the `StateManager`.

The field number is examined to see if it is a declared field or an inherited field. If it is inherited, then the call is delegated to the superclass. If it is declared, then the appropriate `StateManager` `providedXXXField` method is called, which provides the `StateManager` with the value for the field.

If the field is out of range (less than zero or greater than the number of managed fields in the class) then a `JDOFatalInternalException` is thrown.

```
public void jdoCopyFields (Object other, int[] fieldNumbers);
```

This method is called by the `StateManager` to create before images of instances for the purpose of rollback. This method copies the specified fields from the other instance, which must be the same class as this instance, and owned by the same `StateManager`.

If the other instance is not assignment compatible with this instance, then `ClassCastException` is thrown. If the other instance is not owned by the same `StateManager`, then `JDOFatalInternalException` is thrown.

```
protected final void jdoCopyField (<class> other, int fieldNum-
ber);
```

This method is called by the `jdoCopyFields` method to copy the specified field from the other instance. If the field number corresponds to a field in a persistence-capable superclass, this method delegates to the superclass method. If the field is out of range (less than zero or greater than the number of managed fields in the class) then a `JDOFatalInternalException` is thrown.

```
private void writeObject(java.io.ObjectOutputStream out)
    throws java.io.IOException{
```

If no user-written method `writeObject` exists, then one will be generated. The generated `writeObject` makes sure that all persistent and transactional serializable fields are loaded into the instance, by calling `jdoPreSerialize()`, and then the default output behavior is invoked on the output stream.

If the class is serializable (either by explicit declaration or by inheritance) then this code will guarantee that the fields are loaded prior to standard serialization. If the class is not serializable, then this code will never be executed.

Note that there is no modification of a user's `readObject`. During the execution of `readObject`, a new transient instance is created. This instance might be made persistent later, but while it is being constructed by serialization, it remains transient.

20.20 Example class: Employee

The following class definitions for persistence capable classes are used in the examples:

```
package com.xyz.hr;
import javax.jdo.spi.*; // generated by enhancer...
class EmployeeKey {
    int empid;
}
class Employee
    implements PersistenceCapable // generated by enhancer...
{
    Employee boss; // relative field 0
    Department dept; // relative field 1
    int empid; // relative field 2, key field
    String name; // relative field 3
```

20.20.1 Generated fields

```
protected transient javax.jdo.spi.StateManager jdoStateManager =
null;
protected transient byte jdoFlags =
    javax.jdo.spi.PersistenceCapable.READ_WRITE_OK;
// if no superclass, the following:
private final static int jdoInheritedFieldCount = 0;
/* otherwise,
private final static int jdoInheritedFieldCount =
```

```
        <persistence-capable-superclass>.jdoGetManagedFieldCount();
    */
    private final static String[] jdoFieldNames = {"boss", "dept", "empid", "name"};
    private final static Class[] jdoFieldTypes = {Employee.class, Department.class, int.class, String.class};
    private final static byte[] jdoFieldFlags = {
        MEDIATE_READ+MEDIATE_WRITE,
        MEDIATE_READ+MEDIATE_WRITE,
        MEDIATE_WRITE,
        CHECK_READ+CHECK_WRITE
    };
    // if no PersistenceCapable superclass, the following:
    private final static Class jdoPersistenceCapableSuperclass = null;
    /* otherwise,
    private final static Class jdoPersistenceCapableSuperclass = <pc-super>;
    private final static long serialVersionUID = 1234567890L;
    */
```

20.20.2 Generated static initializer

```
static {
    javax.jdo.spi.JDOImplHelper.registerClass (
        Employee.class,
        jdoFieldNames,
        jdoFieldTypes,
        jdoFieldFlags,
        jdoPersistenceCapableSuperclass,
        new Employee());
}
```

20.20.3 Generated interrogatives

```
public final boolean jdoIsPersistent() {
    return jdoStateManager==null?false:
        jdoStateManager.isPersistent(this);
}
public final boolean jdoIsTransactional(){
    return jdoStateManager==null?false:
        jdoStateManager.isTransactional(this);
}
public final boolean jdoIsNew(){
```

```
        return jdoStateManager==null?false:
            jdoStateManager.isNew(this);
    }
    public final boolean jdoIsDirty(){
        return jdoStateManager==null?false:
            jdoStateManager.isDirty(this);
    }
    public final boolean jdoIsDeleted(){
        return jdoStateManager==null?false:
            jdoStateManager.isDeleted(this);
    }
    public final void jdoMakeDirty (String fieldName){
        if (jdoStateManager==null) return;
        jdoStateManager.makeDirty(this, fieldName);
    }
    public final PersistenceManager jdoGetPersistenceManager(){
        return jdoStateManager==null?null:
            jdoStateManager.getPersistenceManager(this);
    }
    public final Object jdoGetObjectId(){
        return jdoStateManager==null?null:
            jdoStateManager.getObjectId(this);
    }
    public final Object jdoGetTransactionalObjectId(){
        return jdoStateManager==null?null:
            jdoStateManager.getTransactionalObjectId(this);
    }
}
```

20.20.4 Generated `jdoReplaceStateManager`

The generated method asks the current `StateManager` to approve the change or validates the caller's authority to set the state.

```
public final synchronized void jdoReplaceStateManager
    (javax.jdo.spi.StateManager sm) {
    // throws exception if current sm didn't request the change
    if (jdoStateManager != null) {
        jdoStateManager = jdoStateManager.replacingStateManager (this,
            sm);
    } else {
        // the following will throw an exception if not authorized
        JDOImplHelper.checkAuthorizedStateManager(sm);
        jdoStateManager = sm;
        this.jdoFlags = LOAD_REQUIRED;
    }
}
```

```
}
```

20.20.5 Generated jdoReplaceFlags

```
public final void jdoReplaceFlags () {  
    if (jdoStateManager != null) {  
        jdoFlags = jdoStateManager.replacingFlags (this);  
    }  
}
```

20.20.6 Generated jdoNewInstance helpers

The first generated helper assigns the value of the passed parameter to the `jdoStateManager` field of the newly created instance.

```
public PersistenceCapable jdoNewInstance(StateManager sm) {  
    // if class is abstract, throw new JDOFatalInternalException()  
    Employee pc = new Employee ();  
    pc.jdoStateManager = sm;  
    pc.jdoFlags = LOAD_REQUIRED;  
    return pc;  
}
```

/* The second generated helper assigns the value of the passed parameter to the `jdoStateManager` field of the newly created instance, and initializes the values of the key fields from the `oid` parameter.

```
*/  
public PersistenceCapable jdoNewInstance(StateManager sm, Object  
oid) {  
    // if class is abstract, throw new JDOFatalInternalException()  
    Employee pc = new Employee ();  
    pc.jdoStateManager = sm;  
    pc.jdoFlags = LOAD_REQUIRED;  
    // now copy the key fields into the new instance  
    jdoCopyKeyFieldsFromObjectId (oid);  
    return pc;  
}
```

20.20.7 Generated jdoGetManagedFieldCount

The generated method returns the number of managed fields in this class plus the number of inherited managed fields. This method is expected to be executed only during class loading of the subclasses.

```
protected static int jdoGetManagedFieldCount () {
```

```
        return jdoInheritedFieldCount + jdoFieldNames.length;
    }
}
```

20.20.8 Generated `jdoGetXXX` methods (one per persistent field)

The access modifier is the same modifier as the corresponding field definition. Therefore, access to the method is controlled by the same policy as for the corresponding field.

```
final static String
jdoGetname(Employee x) {
    // this field is in the default fetch group (CHECK_READ)
    if (x.jdoFlags <= READ_WRITE_OK) {
        // ok to read
        return x.name;
    }
    // field needs to be fetched from StateManager
    // this call might result in name being stored in instance
    StateManager sm = x.jdoStateManager;
    if (sm != null) {
        if (sm.isLoaded (x, jdoInheritedFieldCount + 3))
            return x.name;

        return sm.getStringField(x, jdoInheritedFieldCount + 3,
                                x.name);
    } else {
        return x.name;
    }
}

final static com.xyz.hr.Department
jdoGetdept(Employee x) {
    // this field is not in the default fetch group (MEDIATE_READ)
    StateManager sm = x.jdoStateManager;
    if (sm != null) {
        if (sm.isLoaded (x, jdoInheritedFieldCount + 1))
            return x.dept;
        return (com.xyz.hr.Department)
            sm.getObjectField(x,
                jdoInheritedFieldCount + 1,
                x.dept);
    } else {
        return x.dept;
    }
}
```

```
    }  
}
```

20.20.9 Generated jdoSetXXX methods (one per persistent field)

The access modifier is the same modifier as the corresponding field definition. Therefore, access to the method is controlled by the same policy as for the corresponding field.

```
final static void  
jdoSetName(Employee x, String newValue) {  
    // this field is in the default fetch group  
    if (x.jdoFlags == READ_WRITE_OK) {  
        // ok to read, write  
        x.name = newValue;  
        return;  
    }  
    StateManager sm = x.jdoStateManager;  
    if (sm != null) {  
        sm.setStringField(x,  
            jdoInheritedFieldCount + 3,  
            x.name,  
            newValue);  
    } else {  
        x.name = newValue;  
    }  
}  
  
final static void  
jdoSetDept(Employee x, com.xyz.hr.Department newValue) {  
    // this field is not in the default fetch group  
    StateManager sm = x.jdoStateManager;  
    if (sm != null) {  
        sm.setObjectField(x,  
            jdoInheritedFieldCount + 1,  
            x.dept, newValue);  
    } else {  
        x.dept = newValue;  
    }  
}
```

20.20.10 Generated jdoReplaceField and jdoReplaceFields

The generated `jdoReplaceField` retrieves a new value from the `StateManager` for one specific field based on field number. This method is called by the `StateManager` whenever it wants to update the value of a field in the instance, for example to store values in the instance from the datastore.

This may be used by the `StateManager` to clear fields and handle cleanup of the objects currently referred to by the fields (e.g., embedded objects).

```
public void jdoReplaceField (int fieldNumber) {
    int relativeField = fieldNumber - jdoInheritedFieldCount;
    switch (relativeField) {
        case (0): boss = (Employee)
            jdoStateManager.replacingObjectField (this,
            fieldNumber);
            break;
        case (1): dept = (Department)
            jdoStateManager.replacingObjectField (this,
            fieldNumber);
            break;
        case (2): empid =
            jdoStateManager.replacingIntField (this,
            fieldNumber);
            break;
        case (3): name =
            jdoStateManager.replacingStringField (this,
            fieldNumber);
            break;
        default:
            /* if there is a pc superclass, delegate to it
            if (relativeField < 0) {
                super.jdoReplaceField (fieldNumber);
            } else {
                throw new IllegalArgumentException("fieldNumber");
            }
            */
            // if there is no pc superclass, throw an exception
            throw new IllegalArgumentException("fieldNumber");
    } // switch
}

public final void jdoReplaceFields (int[] fieldNumbers) {
```



```
        for (int i = 0; i < fieldNumbers.length; ++i) {
            int fieldNumber = fieldNumbers[i];
            jdoReplaceField (fieldNumber);
        }
    }
```

20.20.11 Generated `jdoProvideField` and `jdoProvideFields`

The generated `jdoProvideField` gives the current value of one field to the `StateManager`. This method is called by the `StateManager` whenever it wants to get the value of a field in the instance, for example to store the field in the datastore.

```
public void jdoProvideField (int fieldNumber) {
    int relativeField = fieldNumber - jdoInheritedFieldCount;
    switch (relativeField) {
        case (0): jdoStateManager.providedObjectField(this,
            fieldNumber, boss);
            break;
        case (1): jdoStateManager.providedObjectField(this,
            fieldNumber, dept);
            break;
        case (2): jdoStateManager.providedIntField(this,
            fieldNumber, empid);
            break;
        case (3): jdoStateManager.providedStringField(this,
            fieldNumber, name);
            break;
        default:
            /* if there is a pc superclass, delegate to it
            if (relativeField < 0) {
                super.jdoProvideField (fieldNumber);
            } else {
                throw new IllegalArgumentException("fieldNumber");
            }
            */
            // if there is no pc superclass, throw an exception
            throw new IllegalArgumentException("fieldNumber");
    } // switch
}

public final void jdoProvideFields (int[] fieldNumbers) {
    for (int i = 0; i < fieldNumbers.length; ++i) {
```

```
        int fieldNumber = fieldNumbers[i];
        jdoProvideField (fieldNumber);
    }
}
```

20.20.12 Generated jdoCopyField and jdoCopyFields methods

The generated `jdoCopyFields` copies fields from another instance to this instance. This method might be used by the `StateManager` to create before images of instances for roll-back, or to restore instances in case of rollback.

This method delegates to method `jdoCopyField` to copy values for all fields requested. To avoid security exposure, `jdoCopyFields` can be invoked only when both instances are owned by the same `StateManager`. Thus, a malicious user cannot use this method to copy fields from a managed instance to a non-managed instance, or to an instance managed by a malicious `StateManager`.

```
public void jdoCopyFields (Object pc, int[] fieldNumbers){
    // the other instance must be owned by the same StateManager
    // and our StateManager must not be null!
    if (((PersistenceCapable)other).jdoStateManager
        != this.jdoStateManager)
        throw new IllegalArgumentException("this.jdoStateManager !=
other.jdoStateManager");
    if (this.jdoStateManager == null)
        throw new IllegalStateException("this.jdoStateManager ==
null");

    // throw ClassCastException if other class is the wrong class
    Employee other = (Employee) pc;
    for (int i = 0; i < fieldNumbers.length; ++i) {
        jdoCopyField (other, fieldNumbers[i]);
    } // for loop
} // jdoCopyFields

protected void jdoCopyField (Employee other, int fieldNumber) {
    int relativeField = fieldNumber - jdoInheritedFieldCount;
    switch (relativeField) {
        case (0): this.boss = other.boss;
            break;
        case (1): this.dept = other.dept;
            break;
        case (2): this.empid = other.empid;
```

```
        break;
    case (3): this.name = other.name;
        break;
    default: // other fields handled in superclass
// this class has no superclass, so throw an exception
    throw new IllegalArgumentException("fieldNumber");
/* if it had a superclass, it would handle the field as follows:
    super.jdoCopyField (other, fieldNumber);
*/
        break;
    } // switch
} // jdoCopyField
```

20.20.13 Generated writeObject method

If no user-written method `writeObject` exists, then one will be generated. The generated `writeObject` makes sure that all persistent and transactional serializable fields are loaded into the instance, and then the default output behavior is invoked on the output stream.

```
private void writeObject(java.io.ObjectOutputStream out)
    throws java.io.IOException{
    jdoPreSerialize();
    out.defaultWriteObject ();
}
```

20.20.14 Generated jdoPreSerialize method

The generated `jdoPreSerialize` method makes sure that all persistent and transactional serializable fields are loaded into the instance by delegating to the corresponding method in `StateManager`.

```
private final void jdoPreSerialize() {
    if (jdoStateManager != null)
        jdoStateManager.preSerialize(this);
}
```

20.20.15 Generated jdoNewObjectIdInstance

The generated methods create and return a new instance of the object id class.

```
public Object jdoNewObjectIdInstance() {
    return new EmployeeKey();
}
public Object jdoNewObjectIdInstance(String str) {
    return new EmployeeKey(str);
}
```

```
}
```

20.20.16 Generated jdoCopyKeyFieldsToObjectId

The generated methods copy key field values from the `PersistenceCapable` instance or from the `ObjectIdFieldSupplier`.

```
public void jdoCopyKeyFieldsToObjectId (ObjectIdFieldSupplier fs,
Object oid) {
    ((EmployeeKey)oid).empid = fs.fetchIntField (2);
}
public void jdoCopyKeyFieldsToObjectId (Object oid) {
    ((EmployeeKey)oid).empid = empid;
}
```

20.20.17 Generated jdoCopyKeyFieldsFromObjectId

The generated methods copy key fields from the object id instance to the `PersistenceCapable` instance or to the `ObjectIdFieldConsumer`.

```
public void jdoCopyKeyFieldsFromObjectId (ObjectIdFieldConsumer
fc, Object oid) {
    fc.storeIntField (2, ((EmployeeKey)oid).empid);
}
protected void jdoCopyKeyFieldsFromObjectId (Object oid) {
    empid = ((EmployeeKey)oid).empid;
}
} // end class definition
```

21 Interface StateManager

This chapter specifies the `StateManager` interface, which is responsible for managing the state of fields of persistence-capable classes in the JDO environment.

NOTE: This interface is not intended to be used by application programmers. It is for use only by implementations.

21.1 Overview

A class that implements the JDO `StateManager` interface must be supplied by the JDO implementation. There is no user-visible behavior for this implementation; its only caller from the user's perspective is the `PersistenceCapable` class.

21.2 Goals

This interface allows the JDO implementation to completely control the behavior of the `PersistenceCapable` classes under management. In particular, the implementation may choose to exploit the caching capabilities of `PersistenceCapable` or not.

The architecture permits JDO implementations to have a singleton `StateManager` for all `PersistenceCapable` instances; a `StateManager` for all `PersistenceCapable` instances associated with a particular `PersistenceManager` or `PersistenceManagerFactory`; a `StateManager` for all `PersistenceCapable` instances of a particular class; or a `StateManager` for each `PersistenceCapable` instance. This list is not intended to be exhaustive, but simply to identify the cases that might be typical.

Clone support

Note that any of the methods in this interface might be called by a clone of a persistence-capable instance, and the implementation of `StateManager` must disconnect the clone upon detecting it. Disconnecting the clone requires setting the clone's `jdoFlags` to `READ_WRITE_OK`; setting the clone's `jdoStateManager` to null; and then returning from the method as if the clone were transient. For example, in response to `isLoaded`, the `StateManager` calls `clone.jdoReplaceFlags(READ_WRITE_OK); clone.replaceStateManager(null); return true`.

```
package javax.jdo.spi;  
interface StateManager {
```

21.3 StateManager Management

The following methods provide for updating the corresponding `PersistenceCapable` fields. These methods are intended to be called only from the `PersistenceCapable` instance.

It is possible for these methods to be called from a cloned instance of a persistent instance (between the time of the execution of `clone()` and the enhancer-generated reset of the `jdoStateManager` and `jdoFlags` fields). In this case, the `StateManager` is not managing the clone. The `StateManager` must detect this case and disconnect the clone from the `StateManager`. The end result of disconnecting is that the `jdoFlags` field is set to `READ_WRITE_OK` and the `jdoStateManager` field is set to `null`.

```
public StateManager replacingStateManager (PersistenceCapable pc,
    StateManager sm);
```

The current `StateManager` should be the only caller of `PersistenceCapable.replaceStateManager`, which calls this method. This method should be called only when the current `StateManager` wants to set the `jdoStateManager` field to `null` to transition the instance to transient.

The `jdoFlags` are completely controlled by the `StateManager`. The meaning of the values are the following:

0: `READ_WRITE_OK`

any negative number: `READ_OK`

any positive number: `LOAD_REQUIRED`

```
public byte replacingFlags(PersistenceCapable pc);
```

This method is called by the `PersistenceCapable` in response to the `StateManager` calling `jdoReplaceFlags`. The `PersistenceCapable` will store the returned value into its `jdoFlags` field.

21.4 PersistenceManager Management

The following method provides for getting the `PersistenceManager`. This method is intended to be called only from the `PersistenceCapable` instance.

```
public PersistenceManager getPersistenceManager (PersistenceCa-
    pable pc);
```

21.5 Dirty management

The following method provides for marking the `PersistenceCapable` instance dirty:

```
public void makeDirty (PersistenceCapable pc, String fieldName);
```

21.6 State queries

The following methods are delegated from the `PersistenceCapable` class, to implement the associated behavior of `PersistenceCapable`.

```
public boolean isPersistent (PersistenceCapable pc);
```

```
public boolean isTransactional (PersistenceCapable pc);
```

```
public boolean isNew (PersistenceCapable pc);
```

```
public boolean isDirty (PersistenceCapable pc);
```

```
public boolean isDeleted (PersistenceCapable pc);
```

21.7 JDO Identity

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

This method returns the JDO identity of the instance.

```
public Object getTransactionalObjectId (PersistenceCapable pc);
```

This method returns the transactional JDO identity of the instance.

21.8 Serialization support

```
public void preSerialize (PersistenceCapable pc);
```

This method loads all non-transient persistent fields in the `PersistenceCapable` instance, as a precursor to serializing the instance. It is called by the generated `jdoPreSerialize()` method in the `PersistenceCapable` class.

21.9 Field Management

The `StateManager` completely controls the behavior of the `PersistenceCapable` with regard to whether fields are loaded or not. Setting the value of the `jdoFlags` field in the `PersistenceCapable` directly affects the behavior of the `PersistenceCapable` with regard to fields in the default fetch group.

- The `StateManager` might choose to never cache any field values in the `PersistenceCapable`, but rather to retrieve the values upon request. To implement this strategy, the `StateManager` will always use the `LOAD_REQUIRED` value for the `jdoFlags`, and will always return false to any call to `isLoading`.
- The `StateManager` might choose to selectively retrieve and cache field values in the `PersistenceCapable`. To implement this strategy, the `StateManager` will always use the `LOAD_REQUIRED` value for `jdoFlags`, and will return true to calls to `isLoading` that refer to fields that are cached in the `PersistenceCapable`.
- The `StateManager` might choose to retrieve at one time all field values for fields in the default fetch group, and to take advantage of the performance optimization in the `PersistenceCapable`. To implement this strategy, the `StateManager` will use the `LOAD_REQUIRED` value for `jdoFlags` only when the fields in the default fetch group are not cached. Once all of the fields in the default fetch group are cached in the `PersistenceCapable`, the `StateManager` will set the value of the `jdoFlags` to `READ_OK`. This will probably be done during the call to `isLoading` made for one of the fields in the default fetch group, and before returning true to the method, the `StateManager` will call `jdoReplaceFields` with the field numbers of all fields in the default fetch group, and will call `jdoReplaceFlags` to set `jdoFlags` to `READ_OK`.
- The `StateManager` might choose to manage updates of fields in the default fetch group individually. To implement this strategy, the `StateManager` will not use the `READ_WRITE_OK` value for `jdoFlags`. This will result in the `PersistenceCapable` always delegating to the `StateManager` for any change to any field. In this way, the `StateManager` can reliably tell when any field changes, and can optimize the writing of data to the store.

The following method is used by the `PersistenceCapable` to determine whether the value of the field is already cached in the `PersistenceCapable` instance. If it is cached (perhaps during the execution of this method) then the value of the field is returned by the `PersistenceCapable` method without further calls to the `StateManager`.

```
boolean isLoading (PersistenceCapable pc, int field);
```

21.9.1 User-requested value of a field

The following methods are used by the `PersistenceCapable` instance to inform the `StateManager` of a user-initiated request to access the value of a persistent field.

The `pc` parameter is the instance of `PersistenceCapable` making the call; the `field` parameter is the field number of the field; and the `currentValue` parameter is the current value of the field in the instance.

The current value of the field is passed as a parameter to allow the `StateManager` to cache values in the `PersistenceCapable`. If the value is cached in the `PersistenceCapable`, then the `StateManager` can simply return the current value provided with the method call.

```
public boolean getBooleanField (PersistenceCapable pc, int field,
boolean currentValue);
public char getCharField (PersistenceCapable pc, int field, char
currentValue);
public byte getByteField (PersistenceCapable pc, int field, byte
currentValue);
public short getShortField (PersistenceCapable pc, int field, short
currentValue);
public int getIntField (PersistenceCapable pc, int field, int cur-
rentValue);
public long getLongField (PersistenceCapable pc, int field, long
currentValue);
public float getFloatField (PersistenceCapable pc, int field, float
currentValue);
public double getDoubleField (PersistenceCapable pc, int field,
double currentValue);
public String getStringField (PersistenceCapable pc, int field,
String currentValue);
public Object getObjectField (PersistenceCapable pc, int field, Ob-
ject currentValue);
```

21.9.2 User-requested modification of a field

The following methods are used by the `PersistenceCapable` instance to inform the `StateManager` of a user-initiated request to modify the value of a persistent field.

The `pc` parameter is the instance of `PersistenceCapable` making the call; the `field` parameter is the field number of the field; the `currentValue` parameter is the current value of the field in the instance; and the `newValue` parameter is the value of the field given by the user method.

```
public void setBooleanField (PersistenceCapable pc, int field,
boolean currentValue, boolean newValue);
```



```
public void setCharField (PersistenceCapable pc, int field, char
currentValue, char newValue);
public void setByteField (PersistenceCapable pc, int field, byte
currentValue, byte newValue);
public void setShortField (PersistenceCapable pc, int field, short
currentValue, short newValue);
public void setIntField (PersistenceCapable pc, int field, int cur-
rentValue, int newValue);
public void setLongField (PersistenceCapable pc, int field, long
currentValue, long newValue);
public void setFloatField (PersistenceCapable pc, int field, float
currentValue, float newValue);
public void setDoubleField (PersistenceCapable pc, int field, dou-
ble currentValue, double newValue);
public void setStringField (PersistenceCapable pc, int field,
String currentValue, String newValue);
public void setObjectField (PersistenceCapable pc, int field, Ob-
ject currentValue, Object newValue);
```

21.9.3 StateManager-requested value of a field

The following methods inform the StateManager of the value of a persistent field requested by the StateManager.

The `pc` parameter is the instance of `PersistenceCapable` making the call; the `field` parameter is the field number of the field; and the `currentValue` parameter is the current value of the field in the instance.

```
public void providedBooleanField (PersistenceCapable pc, int field,
boolean currentValue);
public void providedCharField (PersistenceCapable pc, int field,
char currentValue);
public void providedByteField (PersistenceCapable pc, int field,
byte currentValue);
public void providedShortField (PersistenceCapable pc, int field,
short currentValue);
public void providedIntField (PersistenceCapable pc, int field, int
currentValue);
public void providedLongField (PersistenceCapable pc, int field,
long currentValue);
public void providedFloatField (PersistenceCapable pc, int field,
float currentValue);
public void providedDoubleField (PersistenceCapable pc, int field,
double currentValue);
public void providedStringField (PersistenceCapable pc, int field,
String currentValue);
public void providedObjectField (PersistenceCapable pc, int field,
Object currentValue);
```

21.9.4 StateManager-requested modification of a field

The following methods ask the `StateManager` for the value of a persistent field requested to be modified by the `StateManager`.

The `pc` parameter is the instance of `PersistenceCapable` making the call; and the `field` parameter is the field number of the field.

```
public boolean replacingBooleanField (PersistenceCapable pc, int
field);
public char replacingCharField (PersistenceCapable pc, int field);
public byte replacingByteField (PersistenceCapable pc, int field);
public short replacingShortField (PersistenceCapable pc, int
field);
public int replacingIntField (PersistenceCapable pc, int field);
public long replacingLongField (PersistenceCapable pc, int field);
public float replacingFloatField (PersistenceCapable pc, int
field);
public double replacingDoubleField (PersistenceCapable pc, int
field);
public String replacingStringField (PersistenceCapable pc, int
field);
public Object replacingObjectField (PersistenceCapable pc, int
field);
```

22 JDOPermission

A permission represents access to a system resource. For a resource access to be allowed for an applet (or an application running with a security manager), the corresponding permission must be explicitly granted to the code attempting the access.

The `JDOPermission` class provides a marker for the security manager to grant access to a class to perform privileged operations necessary for JDO implementations.

There are three JDO permissions defined:

- `setStateManager`: this permission allows an instance to manage an instance of `PersistenceCapable`, which allows the instance to access and modify any fields defined as persistent or transactional. This permission is similar to but allows access to only a subset of the broader `ReflectPermission("suppressAccessChecks")`. This permission is checked by the `PersistenceCapable.replaceStateManager` method.
- `getMetadata`: this permission allows an instance to access the metadata for any registered `PersistenceCapable` class. This permission allows access to a subset of the broader `RuntimePermission("accessDeclaredMembers")`. This permission is checked by the `JDOImplHelper.getJDOImplHelper` method.
- `closePersistenceManagerFactory`: this permission allows a caller to close a `PersistenceManagerFactory`, thereby releasing resources. This permission is checked by the `close()` method of `PersistenceManagerFactory`.

Use of `JDOPermission` allows the security manager to restrict potentially malicious classes from accessing information contained in instances of `PersistenceCapable`.

A sample policy file entry granting code from the `/home/jdoImpl` directory permission to get metadata, manage `PersistenceCapable` instances, and close `PersistenceManagerFactory` instances is

```
grant codeBase "file:/home/jdoImpl/" {
    permission javax.jdo.spi.JDOPermission "getMetadata";
    permission javax.jdo.spi.JDOPermission "setStateManager";
    permission javax.jdo.spi.JDOPermission
        "closePersistenceManagerFactory";
};
```

23 JDO Query BNF

23.1 Grammar Notation

The grammar notation is taken from the Java Language Specification:

- Terminal symbols are shown in **bold** in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.
- Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines.
- The suffix “*opt*”, which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it.
- When the words “one of” follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition.

23.2 Parameter Declaration

This section describes the syntax of the `declareParameters` argument.

DeclareParameters:

Parameters , *opt*

Parameters:

Parameter

Parameters , *Parameter*

Parameter:

Type Identifier

23.3 Variable Declaration

This section describes the syntax of the `declareVariables` argument.

DeclareVariables:

Variables ;_{opt}

Variables:

Variable

Variables ; Variable

Variable:

Type Identifier

23.4 Import Declaration

This section describes the syntax of the `declareImports` argument.

DeclareImports:

ImportDeclarations ;_{opt}

ImportDeclarations:

ImportDeclaration

ImportDeclarations ; ImportDeclaration

ImportDeclaration:

import *Name*

import *Name.**

23.5 Ordering Specification

This section describes the syntax of the `setOrdering` argument.

SetOrdering:

OrderSpecifications ,_{opt}

OrderSpecifications:

OrderSpecification

OrderSpecifications , OrderSpecification

OrderSpecification:

Expression **ascending**

Expression **descending**

23.6 Filter Expression

This section describes the syntax of the `setFilter` argument.

Basically, the query filter expression is a Java boolean expression, where some of the Java expression are not permitted. Specifically, pre- and post- increment and decrement (`++` and `--`), shift (`>>` and `<<`) and assignment expressions (`+=`, `-=`, etc.) are not permitted.

The description is bottom-up, i.e. the last rule `Expression` is the root of the filter expression syntax.

Please note, the grammar allows arbitrary method calls (`MethodInvocation`), where JDO permits calls only to the methods `contains()`, `isEmpty()`, and a number of `String` methods. This restriction cannot be expressed in terms of the syntax and has to be ensured by a semantic check.

Primary:

Literal

this

(*Expression*)

FieldAccess

MethodInvocation

ArgumentList:

Expression

ArgumentList , Expression

FieldAccess:

Primary . Identifier

MethodInvocation:

Primary . Identifier (ArgumentList_{opt})

PostfixExpression:

Primary

Name

UnaryExpression:

+ UnaryExpression

- UnaryExpression

UnaryExpressionNotPlusMinus

UnaryExpressionNotPlusMinus:

PostfixExpression

~ UnaryExpression

! UnaryExpression

CastExpression

CastExpression:

(Type) UnaryExpression

MultiplicativeExpression:

UnaryExpression

*MultiplicativeExpression * UnaryExpression*

MultiplicativeExpression / UnaryExpression

AdditiveExpression:

MultiplicativeExpression

AdditiveExpression + MultiplicativeExpression

AdditiveExpression - MultiplicativeExpression

RelationalExpression:

AdditiveExpression

RelationalExpression < AdditiveExpression

RelationalExpression > AdditiveExpression

RelationalExpression <= AdditiveExpression

RelationalExpression >= AdditiveExpression

EqualityExpression:

RelationalExpression

EqualityExpression == RelationalExpression

EqualityExpression != RelationalExpression

AndExpression:

EqualityExpression

AndExpression & EqualityExpression

InclusiveOrExpression:

AndExpression

InclusiveOrExpression | AndExpression

ConditionalAndExpression:

InclusiveOrExpression

ConditionalAndExpression && InclusiveOrExpression

ConditionalOrExpression:

ConditionalAndExpression

ConditionalOrExpression || *ConditionalAndExpression*

Expression:

ConditionalOrExpression

23.7 Types

This section describes a type specification, used in a parameter or variable declaration or in a cast expression.

Type

PrimitiveType

Name

PrimitiveType:

NumericType

boolean

NumericType:

IntegralType

FloatingPointType

IntegralType: one of

byte short int long char

FloatingPointType: one of

float double

23.8 Literals

A literal is the source code representation of a value of a primitive type, the `String` type, or the null type. Please refer to the Java Language Specification for the lexical structure of `IntegerLiterals`, `FloatingPointLiterals`, `CharacterLiterals` and `StringLiterals`.

IntegerLiteral: ...

FloatingPointLiteral: ...

BooleanLiteral: one of

true false

CharacterLiteral: ...

StringLiteral: ...

NullLiteral:

null

Literal:

IntegerLiteral

FloatingPointLiteral

BooleanLiteral

CharacterLiteral

StringLiteral

NullLiteral

23.9 Names

A name is a possibly qualified identifier. Please refer to the Java Language Specification for the lexical structure of an identifier.

Name:

Identifier

QualifiedName

QualifiedName:

Name . *Identifier*

24 Items deferred to the next release

This chapter contains the list of items that were raised during the development of JDO but were not resolved.

24.1 Nested Transactions

Define the semantics of nested transactions.

24.2 Savepoint, Undosavepoint

Related to nested transactions, savepoints allow for making changes to instances and then undoing those changes without making any datastore changes. It is a single-child nested transaction.

24.3 Inter-PersistenceManager References

Explain how to establish and maintain relationships between persistent instances managed by different `PersistenceManagers`.

24.4 Enhancer Invocation API

A standard interface to call the enhancer will be defined.

24.5 Prefetch API

A standard interface to specify prefetching of instances by policy will be defined. The intended use it to allow the application to specify a policy whereby instances of persistence capable classes will be prefetched from the datastore when related instances are fetched. This should result in improved performance characteristics if the prefetch policy matches actual application access patterns.

24.6 BLOB/CLOB datatype support

JDO implementations can choose to implement mapping from `java.sql.Blob` datatype to byte arrays, and `java.sql.Clob` to String or other java type; but these mappings are not standard, and may not have the performance characteristics desired.

24.7 Managed (inverse) relationship support

In order for JDO implementations to be used for container managed persistence entity beans, relationships among persistent instances need to be explicitly managed. See the EJB Specification 2.0, sections 9.4.6 and 9.4.7 for requirements. The intent is to support these semantics when the relationships are identified in the metadata as inverse relationships.

24.8 Case-Insensitive Query

Use of `String.toLowerCase()` as a supported method in query filters would allow case-insensitive queries.

24.9 String conversion in Query

Supported String constructors `String(<integer expression>)` and `String(<floating-point expression>)` would make queries more flexible.

24.10 Read-only fields

Support (probably marking the fields in the XML metadata) for read-only fields would allow better support for databases where modification of data elements is proscribed. The metadata annotation would permit earlier detection of incorrect modification of the corresponding fields.

24.11 Enumeration pattern

The enumeration pattern is a powerful technique for emulating enums. The pattern in summary allows for fields to be declared as:

```
class Foo {
    Bar myBar = Bar.ONE;
    Bar someBar = new Bar("illegal"); // doesn't compile
}
class Bar {
    private String istr;
    private Bar(String s) {
        istr = s;
    }
    public static Bar ONE = new Bar("one");
    public static Bar TWO = new Bar("two");
}
```

The advantage of this pattern is that fields intended to contain only certain values can be constrained to those values. Supporting this pattern explicitly allows for classes that use this pattern to be supported as persistence-capable classes.

24.12 Non-static inner classes

Allow non-static inner classes to be persistence-capable. The implication is that the enclosing class must also be persistence-capable, and there is a one-many relationship between the enclosing class and the inner class.

24.13 Projections in query

Currently the only return value from a JDOQL query is a Collection of persistent instances. Many applications need values returned from queries, not instances. For example, to properly support EJBQL, projections are required. One way to provide projections is to model what EJBQL has already done, and add a method setResult (String projection) to javax.jdo.Query. This method would take as a parameter a single-valued navigation expression. The result of execute for the query would be a Collection of instances of the expression.

24.14 LogWriter support

Currently, there is no direct support for writing log messages from an implementation, although there is a connection factory property that can be used for this purpose. A future revision could define how an implementation should use a log writer.

24.15 New Exceptions

Some exceptions might be added to more clearly define the cause of an exception. Candidates include JDODuplicateObjectIdException, JDOClassNotPersistenceCapableException, JDOExtentNotManagedException, JDOConcurrentModificationException, JDOQueryException, JDOQuerySyntaxException, JDOUnboundQueryParameterException, JDOTransactionNotActiveException, JDODeletedObjectFieldAccessException.

24.16 Distributed object support

Provide for remote object graph support, including instance reconciliation, relationship graph management, instance insertion ordering, etc.

24.17 Object-Relational Mapping

Extend the current xml metadata to include optional O/R mapping information. This could include tables to map to classes, columns to map to fields, and foreign keys to map to relationships.

Other O/R mapping issues include sequence generation for primary key support.

Appendix A: References

- [1] **Enterprise JavaBeans (EJB) specification:**
<http://java.sun.com/products/ejb/docs.html>
- [2] **Java Transaction API (JTA) specification - version 1.0**
<http://java.sun.com/products/jta/>
- [3] **Java 2 Platform Enterprise Edition (J2EE), Platform specification:**
<http://java.sun.com/j2ee/docs.html>
- [4] **Java 2 Platform Enterprise Edition (J2EE), Connector Architecture:**
<http://java.sun.com/j2ee/apidocs/>
<http://java.sun.com/j2ee/download.html#connectorspec>

Appendix B: Design Decisions

This appendix outlines some of the design decisions that were considered and not taken, along with the rationale.

B.1 Enhancer

The enhancer could generate code that would delegate to the associated `StateManager` every access (read or write) for every field. This design was rejected because of several factors.

- Code bloat: the enhanced code would add an extra method call to every access to a persistent field.
- Performance: the calls to the `StateManager` would add extra cycles to every access to a persistent field, even if the field were already fetched into the persistent instance.

The enhancer could require complete metadata descriptions for all persistence-capable classes and persistent and transactional fields, and further require that all classes be available during enhancement of any class.

This would allow the enhancer to generate the most efficient code, but imposes an extra burden on the user to keep the metadata and class definition absolutely in sync. If a field were declared in a class after the metadata was defined, the user would have to update the metadata to add the new field.

Requiring access to all classes during enhancement of any class was also seen as an extra burden on the user, who would have to execute the enhancement in an environment that did not necessarily reflect the runtime environment. There is also a performance penalty and additional complexity for the enhancer.

The decision that was taken was that the enhancer must be able to determine the persistence-modifier (persistent or none) from the Java modifiers and type of a field. Further, the information needed to enhance a class is only the class file for the class being enhanced, plus the metadata for the class and classes directly reachable (via references or inheritance) from the class.

The java byte codes generated in a class for a field in another class do not contain much information about the modifiers (final or transient) of the field. They do have the field name and the field type, and whether the field is static. There is an implied access control that permits the generated access (package, protected, or public) but no distinction among the choices.

Therefore, a field that is not declared in the metadata must be enhanced to generate an accessor and mutator even though the field is not persistent. For example, for a final int field declared in a class, the field is not persistent, so it is not included in the list of persistent/transactional fields, but an accessor is generated for it. This accessor will be used only by other classes' accesses, and access will not be mediated (the `StateManager` will never be called). Accesses within the class are not enhanced.

B.2 PersistenceCapable

The `PersistenceCapable` interface could be eliminated entirely in favor of having all interrogatives operate via the `PersistenceManager`, not directly on the JDO instance. This would make the JDO instance entirely user-written. However, the impact would be that to find out which `PersistenceManager`, if any, was responsible for the JDO instance, a new

singleton would have to be provided. The singleton would have to register all PersistenceManager instances and ask each if it managed a specific JDO instance.

This was deemed too complex to manage, as well as too slow to find simple information that should be easily available.

B.3 Collection Factory

The collection factory could be defined as methods on PersistenceManager or as methods on a separate interface. Also, a single method that takes a type, or multiple methods, one for each type could be defined.

The decision was taken to define two methods on PersistenceManager based on the requirement to create an instance of a collection based on the type of an existing instance. This operation would be complex if individual methods were used, one per type.

A convenience interface can easily be created using the defined methods.

Appendix C: Revision History

This appendix outlines the significant changes during the evolution of the specification.

C.1 Changes since Draft 0.1

- Added Appendix for revision history
- Added Appendix for design decisions not taken

C.2 Changes since Draft 0.2

- Changed the description for the persistent state (cached non-transactional values)
- Added JDO instance state transition diagram and descriptions of state transitions.
- Enhanced description of non-datastore JDO identity.
- Added persistent-new-dirty and persistent-new-clean states to the life cycle.
- Removed the `checkpoint` method from the `Transaction` interface. This functionality is now done by the `TRANSACTION_RETAIN_VALUES` `Transaction` flag.
- Added `jdoCopy` to the `PersistenceCapable` interface.
- Added `Query` interface.

C.3 Changes since Draft 0.3

- Changed `Query` signatures for `setVars` and `setParams`.
- Changed all “set” `Query` signatures to return `void` instead of “`Query`”.
- Added description of key (JDO identity) change semantics.
- Added life cycle description for `deletePersistent`, a new interrogatory `jdoIsDeleted`, and two new states `persistent-new-deleted` and `persistent-deleted`.
- Added Chapter 6 Persistent Object Model, which specifies the field types for persistent fields, including the required `Collection` types.
- Added descriptions of enhancement to Chapter 13 JDO Enhancer, including serialization, cloning, and reflection.
- Added multiple object versions of `makePersistent`, `makeTransactional`, `makeNontransactional`.

C.4 Changes since Draft 0.4

C.4.1 PersistenceManager

- Removed `flush` and `postCompletion` from the API.
- Changed `refresh` to indicate it is effective only in optimistic transactions.
- Removed `getFlags` and `setFlags`, substituting `getXXX` and `setXXX` for all options.
- Added `getProperties`, which returns `VendorName`, `VersionNumber`, etc.
- Added `get/setUserObject`, which allow a user-specified object to be remembered by the `PersistenceManager`.
- Required the implementation to support `PersistenceManagerFactory` and specified the interface for it.

Associated the concept of Extent with `makePersistent` and `deletePersistent`. Only classes with a managed Extent can be parameters of these methods.

Added `getObjectIdClass` to allow the application to get the `ObjectId` class for a class.

C.4.2 Query

Added `newQuery` (Class `cls`, String `filter`).

Changed signature of `compile` to return `void`. This is not required to do anything but validate query elements.

Made the `Query` implementation class serializable. A serialized and restored query instance can be bound to a `PersistenceManager` by `newQuery` (Object).

Removed `execute` methods with four, five, and six parameters.

Allowed Date comparisons for equality and range queries.

Allowed String comparisons for equality and range queries.

Added “`this`” as a valid keyword in filters.

Added a query option to indicate faster queries that don’t execute the filter on cached instances.

Clarified that portable applications require all variables to be scoped by a `contains` clause.

Defined that variables not scoped by a `contains` clause are scoped by the Extent of the class.

C.4.3 Object Model

Changed the name of “Tracked SCO” to “SCO”.

Required a transaction to be in effect to execute `makePersistent` and `deletePersistent`.

Allowed an implementation to treat all reference types as First Class Objects.

Sharing of SCOs is permitted but the semantics are not guaranteed to be portable.

C.4.4 Life Cycle

Removed state `persistent-new-clean` and changed the name of `persistent-new-dirty` to `persistent-new`.

Updated life cycle state diagram to simplify state transition descriptions.

Added section describing optimistic transaction state changes.

C.4.5 PersistenceCapable

Removed methods `jdoIsReadReady` and `jdoIsWriteReady`. None of the application’s business, these.

Changed the semantics of `jdoIsTransactional` to return `false` if an instance is read in an optimistic transaction. In an optimistic transaction, only new, deleted, modified instances and instances made transactional return `true`.

Added `jdoGetPersistenceManager`, `jdoGetObjectId`, and `jdoMakeDirty`.

C.5 Changes since Draft 0.5

Clarified `NontransactionalRead`, `Optimistic`, and `RetainValues` flag dependencies.

Added a table and diagrams of life cycle transitions.

Changed datastore `ObjectId` to allow primitive wrapper classes to be used.

Added failed object array and methods to JDOException, JDOCanRetryException, JDO-DataStoreException, and JDOUserException.

Added a Chapter on Application Portability Guidelines.

Added a Chapter on XML Metadata.

Added two collection factories to PersistenceManager.

Added connection factory to PersistenceManagerFactory.

C.6 Changes since Draft 0.6 (Participant Review Draft)

Updated life cycle table to match transition descriptions for persistent-nontransactional instances. Clarified that all data accessed while a datastore transaction is in progress will be transactional.

Added a discussion on inheritance issues for persistence capable classes.

Added class JDOHelper with static methods to avoid calling JDO specific methods on PersistenceCapable classes.

Added a discussion on using the life cycle methods of PersistenceManager to clarify that the correct method must be called if an instance that implements a Collection interface is to be a parameter.

Query use of operator = was extended to include pre- and post-increment and -decrement operators.

Query variables need not be unique; if they need to be unique, then uniqueness can be specified with an additional query term.

Query examples were clarified as to their intent.

The terms persistent, non-persistent, transient were made consistent throughout the document. “Persistent field” and “non-persistent field” refer to fields as declared in the JDO metadata. “Transient field” refers to the field modifiers (orthogonal to persistent/non-persistent) and “transient instance” refers to an instance of a persistence capable class that is not persistent. “Persistent instance” refers to an instance of a persistence capable class that is persistent.

Derived fields were removed. These fields were supposed to be non-persistent fields whose values depended on values of persistent fields. For example, age depends on birth-date. The application will have to have a method age() instead of an instance variable age.

Transactional non-persistent fields were added. These fields have their values saved and restored during rollback transitions along with persistent fields.

More details were added on use of JDO in the EJB environment.

C.7 Changes since Draft 0.7

Binary compatibility table was added to 2.1.1.

Optional features were added to Portability Guidelines.

Section 5.5.2 was clarified to require that the JDO identity instance can be obtained immediately after the transition from transient to persistent-new.

The treatment of marking fields dirty for hidden fields was changed.

A table of arithmetic operators was added to the Query section.

C.8 Changes since Draft 0.8

Query filter defaults to “true” if not specified.

Added `java.lang.BigInteger`, `java.lang.BigDecimal` to object model.

Added cast operator (class) to query filter syntax.

Added bitwise invert operator to query filter syntax.

Added unary + to query filter syntax.

Added parentheses to query filter syntax.

Added String methods `beginsWith` and `endsWith` to query filter syntax.

Added chapter for `StateManager` interface.

Rewrote entire chapter on Reference Enhancer.

Updated `PersistenceCapable` interface to match Reference Enhancer.

Removed `PersistenceManager.setObjectId`.

Updated XML to conform to `xml4j` DOM and Apache/Xerces verifying parsers.

Added second-class XML attribute to field element.

Added null-value XML attribute to field element. This attribute specifies the behavior of the runtime system when a null-valued field mapped to a non-nullable datastore element is stored. The user can choose to throw an exception or to convert the null value to a default datastore value.

Changed the description of life cycle states and enhancer to indicate that primary key field access is always permitted, regardless of the life cycle state.

Added Extent chapter. The Extent interface was defined to be the result type of `PersistenceManager.getExtent`. The interface does not have the methods of `Collection`, so it can be used only for iteration or for specifying the candidate instances for Query.

Fields in an inherited class may not be managed by a persistence capable class. It is a future objective to allow a class to manage the state of inherited fields if it directly derives from a non-persistence capable class.

Clarified the behavior of null parameters in calls to `PersistenceManager`. Null values are permitted as parameters for `PersistenceCapable` instances, and permitted as elements of `Collection` and `Object[]` parameters, but are not permitted as parameters for `Collection` and `Object[]`.

Added `JDOPermission` class to allow security management to enable jdo implementations without requiring `ReflectPermission`, which is too permissive.

C.9 Changes since Draft 0.9

Updated XML Metadata

- Added xml version number
- Changed definition of class element to allow multiple field, vendor elements
- Added jdo element, which contains multiple package elements
- Added key-type to field element for Map types.
- Changed key-type in class element to identity-type
- Changed key-class in class element to `objectid-class`
- Added inverse to field element for managed relationships

- Added has-extent to class element

Fixed missing “static” in generated `jdoInheritedFieldCount`.

Fixed `jdoGetXXX/jdoSetXXX` in enhanced code for non-dfg fields. Transient instances would have thrown null pointer exception.

Fixed missing generated method in `PersistenceCapable`: `PersistenceCapable jdoNewInstance(StateManager sm)`

Fixed the reference to the Connector Architecture in Appendix A.

Updated ordering to include expressions and restrict the types of ordering expressions to primitives except boolean, wrappers except `Boolean`, `BigDecimal`, `BigInteger`, and `Date`.

Removed bitwise AND, OR, and XOR from query operators.

Changed signatures of `PersistenceManager` methods `getObjectById` and `getTransactionalInstance` to include a boolean flag indicating whether to validate that the instance exists in the datastore.

Clarified that `getObjectId` returns the identity as of the beginning of the transaction, in case the identity is being modified in the transaction.

C.10 Changes since draft 0.91

Changed `xml` has-extent to `requires-extent`

Corrected the signature of `replacingIntField` in `StateManager`.

Corrected the example code generated for `PersistenceCapable jdoReplaceField`.

Corrected the name of the `verify` parameter to `validate` in the signature of `getObjectById`.

Removed `getTransactionalInstance` in favor of overloading the meaning of `getObjectById`.

Changed the requirement to expose the hollow state to the application. A JDO implementation might perform a state transition of a hollow instance as if the application had read a field.

Changed inheritance rules to allow non-persistence-capable classes to have persistence-capable superclasses and subclasses.

Corrected the description of the field name in the `markDirty` method so an unqualified name refers to the field in the most-derived class.

Corrected the signature of the `newInstance` method in `JDOHelper` to return `Object`.

Updated the instance callback description to include the rationale and environment for callbacks.

Updated `makePersistent` and `deletePersistent` to remove the restriction that the class of the instances must have an `Extent`.

The behavior of failing instances in the life cycle methods was clarified to specify that all instances will be attempted, and all failing instances will be included in the exception.

The `newCollectionInstance` was modified to include an `initialContents` parameter.

A new method `newMapInstance` was created to allow construction of a second class map instance.

Optimistic transaction management was clarified to specify that instances accessed during an optimistic transaction are not enlisted in any datastore transaction until commit.

The ordering specification was modified to include `String`.

The `isEmpty` method was added to the allowed `Collection` methods in query.

The treatment of null-valued collection fields was specified to be identical to fields containing empty collections.

Specified the behavior of the iterator of an Extent if there are deleted or newly persistent instances in the Extent.

The chapter on EJB has been substantially redone.

Exceptions were updated as to the contents of the failed object array.

The meaning of JDOHelper.getObjectId versus PersistenceManager.getObjectId was clarified with regard to change of identity within a transaction.

Fixed (removed) all references to reference parameter in StateManager.

Changed interface in PersistenceCapable for creating new instances, registering the PersistenceCapable class with the runtime, and managing minimal “reflective” metadata for the runtime (managed field names and types).

Added chapters for JDOHelper and JDOImplHelper.

C.11 Changes since draft 0.92

PersistenceManager methods that take a collection or array of instances have been changed to include All in their names.

Text throughout the document has been clarified to refer to the specific exception thrown.

Corrected sample code generated by the enhancer.

Added PersistenceManagerFactory methods getPersistenceManager(String userid, String password).

Static fields for values of jdoFlags were added to the PersistenceCapable interface.

A new ELEMENT array was added to the XML metadata to specify for array types whether the elements are embedded or not.

Clarified the possible treatment of jdoFlags by the StateManager, and the handling of is-Loaded.

Added methods PersistenceManager.getTransactionObjectId, PersistenceCapable.jdoGetTransactionObjectId, and JDOHelper.getTransactionObjectId to cover the case of changing primary key in a transaction.

Changed the requirement for a compliant implementation to support all Collection types. The behavior of all Collection types is specified, but only Collection, Set, and HashSet are required.

Clarified the semantics of getObjectId with the validate flag set to true when the instance is in the cache, for the cases of transactional v. nontransactional instances.

Changed failedObjectArray to failedObject, and nestedException to nestedExceptionArray in JDOException.

C.12 Changes since draft 0.93

Removed the requirement for application identity key classes to implement equals for all object types that include the correct name and type fields.

Changed the state transition of persistent-deleted to be unchanged by refresh.

Added a generated constructor jdoNewObjectIdInstance to facilitate key class handling.

Added a generated constructor jdoNewInstance (StateManager sm, Object oid) to facilitate key class handling.

Added generated `jdoCopyKeyFieldsToObjectId` methods to facilitate key class handling.

Added nested interface `ObjectIdFieldManager` to facilitate key class handling.

Added `PersistenceManagerFactory` properties `ConnectionFactory2` and `ConnectionFactory2Name` for application server optimistic transaction support.

Added `loadFactor` to the `newCollectionInstance` method.

Clarified handling of `getObjectId`, `getObjectById`, and `validate`.

Added methods `close(Iterator)` and `closeAll()` to `Extent`.

Added methods `close (Object queryResult)` and `closeAll()` to `Query`.

Updated EJB chapter to clarify life cycle changes.

Removed `inverse` from XML metadata.

Corrected some code examples in reference enhancer.

Added methods to support different query languages: `PersistenceManager.newQuery (String language, Object query)` and `Set supportedQueryLanguages()`.

Added nested extensions, and package extensions to `xml`.

C.13 Changes since draft 0.94

Added `PersistenceManager` and `PersistenceManagerFactory` methods to support the `Multithreaded` property. This property indicates that the application is multithreaded (multiple threads will access instances managed by the `PersistenceManager`).

Removed the `PersistenceCapable` constructor that takes `StateManager` as an argument. The helper methods `newInstance` will use the default constructor instead, and will create protected default constructor if none exists.

Removed `jdoVersionUID` and replaced it with explicit `byte[] jdoFieldFlags` and `Class jdoPersistenceCapableSuperclass`.

Added static fields to define values for `jdoFieldFlags` elements.

Added a chapter on `JDOPermission`.

Added optional extension element to `xml` elements `array`, `collection`, and `map`.

Added `Multithreaded` property to `PersistenceManager`, which indicates whether the `PersistenceManager` must synchronize accesses from multiple application threads.

Added `allowNulls` parameter to `PersistenceManager.newInstance`.

Changed the name of the method `getJDOImplHelper` to `getInstance`.

Clarified the handling of abstract classes, which might be `PersistenceCapable` (for the benefit of concrete subclasses).

Removed the requirement for implementations to track modifications made to arrays.

Removed method `getProperties` from `PersistenceManager`. This method now is in `PersistenceManagerFactory` only.

Removed `supportedQuery` from `PersistenceManager`. This method has been replaced by `supportedOptions`, from which supported query languages should be available.

Added a method `supportedOptions` to `PersistenceManagerFactory` for the application to determine which optional features are supported by an implementation.

Added query BNF chapter.

C.14 Changes since draft 0.95 (Proposed Final Draft)

Defined the term “Managed Fields” to mean persistent or transactional fields.

Clarified the treatment of non-managed identity if multiple instances are changed or deleted.

Removed the requirement that a transaction be active to make an instance transactional or nontransactional.

Reorganized the State Transitions table to indicate that some state transitions are impossible (e.g. without a transaction active, there can be no new instances).

Clarified the requirement for a no-args constructor in PersistenceCapable classes and superclasses.

Fixed bug in PersistenceCapable.replaceStateManager code generation.

Removed properties minPool, maxPool, msWait, and ConnectionDriverName from the interface. These can be specified by PersistenceManagerFactory implementations as needed.

Reorganized sections 20.14 through 20.16 for clarity.

Changed jdoFieldFlags to be independent flags, allowing for identification of non-transient (serializable) fields.

Reworded the transaction synchronization sections for clarity.

Reworded the optimistic transaction section for clarity.

Modified the String concatenation operator (+) to allow only String + String, not String + primitive.

Clarified that String comparisons are lexicographical (not Locale-specific).

Added descriptions of JDOUserException for transaction not active and object deleted.

C.15 Changes since draft 0.96

Changed to specify that String comparisons in queries are based on an ordering not specified by JDO, allowing for locale-specific orderings by JDO implementations.

Added a portability requirement for object id classes to have a toString() method and a public constructor that takes a String argument. Added newObjectIdInstance (Class, String) to PersistenceCapable, jdoNewObjectIdInstance(String) to PersistenceCapable and newObjectIdInstance(Class, String) to JDOImplHelper.

Split PersistenceCapable.ObjectIdFieldManager into two interfaces: PersistenceCapable.ObjectIdFieldSupplier to supply values and PersistenceCapable.ObjectIdFieldConsumer to receive values.

Added the ability to construct a PersistenceManagerFactory from a Properties instance containing keys and values of properties. Added a convenience method to JDOHelper getPersistenceManagerFactory(Properties) to call the method in the implementation class.

Changed SCO factory name to newTrackedInstance, and removed the simultaneous setting of the field value in the persistence-capable instance. The user must assign the newly created instance to a field directly.

Added a parameter to newTrackedInstance to allow the user to specify a comparator for Collection or Map.

Modified the behavior of makePersistent with regard to reachable instances. The newly reachable instances have the characteristics of persistent-new until transaction end, at which time they either become persistent or revert to transient.

Made support for application changes to application object identity an optional feature.

Methods `retrieve` and `retrieveAll` were added to `PersistenceManager` to allow the application to give the implementation a hint that the instances are going to be used by the application, and the implementation can perform some optimized fetching of the instances.

Introduced the notion of provisional persistence. Instances that are reachable by persistent fields from instances made persistent become provisionally persistent. They behave like persistent instances until commit, at which time if they are no longer reachable from persistent instances they revert to transient.

Type-import-on-demand (`import <package-name>.*`) has been added to query `declareImports`. The Java rules for determining the package for an unqualified name are followed by query.

The new `Query` methods that take both `Extent` and `Class` have been changed to eliminate the `Class` argument. The `Class` is taken from the `Extent`.

The Reference Enhancement chapter was reorganized to make it easier to determine: changes to `PersistenceCapable` root classes; changes to non-root classes; and changes to non-`PersistenceCapable` classes.

Changed the signatures of `StateManager` interface methods to take `PersistenceCapable` as the first argument, to avoid a cast operation.

Defined a new method to be enhanced into the least-derived `PersistenceCapable` class to handle copying key fields from oid into the instance: `jdoCopyKeyFieldsFromObjectId` (`Object oid`).

Removed that `makeDirty` in `JDOHelper` throws an exception in the case that the instance is not transient and the field is not managed. This is only one case that throws an exception; the other cases silently ignore the condition. To be consistent, this condition will also silently return.

C.16 Changes since draft 0.97

Clarified comparisons in JDOQL for wrapped types and promotion of numeric types.

Made static method `getPersistenceManagerFactory(Properties)` mandatory for JDO implementations.

Added `PersistenceManagerFactory` property `ConnectionDriverName`.

Added vendor-specific global configuration data in the first part of a `XXX.jdo` file. For this, the DTD was changed from `<!ELEMENT jdo (package)+>` to `<!ELEMENT jdo (package)+(extension)*>`.

Clarified that the class of a persistent instance must be preserved, unless some outside change is made to the datastore.

Clarified that parameters to query must be persistent, associated with the same `PersistenceManager` as the `Query`.

Clarified that for portability, the instances in a candidate collection must be persistent, associated with the same `PersistenceManager` as the `Query`.

Changed the semantics of `retrieve` and `retrieveAll` to require that the `PersistenceManager` load all fields of the parameter instances, so a subsequent call to `makeTransient` can operate on a valid instance (all persistent fields loaded).

Added description of class loaders to the `PersistenceManager` chapter 12.5.

Clarified that there are no default values for flags in `getPersistenceManager`.

Added transaction flag `restoreValues`, which determines the treatment of persistent instances at transaction rollback.

Changed the specification of application identity key classes to require (instead of recommend) that the class override the `toString` method and provide a public constructor that takes only a `String` parameter.

Clarified query comparisons for persistent and transient parameters and candidate instances.

C.17 Changes since Approved Draft

Changed 3.2.1 to correct the interface name from `javax.jdo.PersistenceCapable` to `javax.jdo.spi.PersistenceCapable`.

Fixed typo in 5.5.6. Changed “The instance loses its JDO Identity and its association with the `PersistenceManager`.” to “The instance retains its JDO Identity and its association with the `PersistenceManager`.”

In 5.4.1 changed the wording regarding field types of application identity key fields to require portable applications to use only primitive, `String`, `Date`, and `Number` types.

In 5.4.1 added a restriction that application object id instances must not have any key fields with a value of null.

Added to 5.6.1 that the `PersistenceManager` must not hold a strong reference to a persistent-nontransactional instance, so that it may be garbage collected.

In 5.8, clarified that a before image might be created on update depending on the implementation of optimistic verification.

Corrected table 2 for rollback entries; changed the flag that affects the operation from `retainValues` to `restoreValues`.

In Figure 13 Note 23, fixed “A persistent-dirty instance transitions to persistent-nontransactional... at rollback when `RestoreValues` set to true.”

In Figure 13 Note 18 fixed from “The instance is cleared of values.” to “No changes are made to the values.”

Clarified 6.3 to discuss the treatment of Second Class Objects embedded in First Class Objects. SCO instances of `PersistenceCapable` types have no standard treatment.

In 8.5, fixed missing property `javax.jdo.option.ConnectionDriverName` in `JDOHelper` list of standard properties for `getPersistenceManagerFactory`.

Added new section 9.5 for new security checking for `StateManager`. The new authorization strategy does not require that the persistence-capable classes be authorized for `JDOPermission(“setStateManager”)`.

Fixed 10.3 the description of `jdoPreClear` does not include deleted instances, as these instances do not transition to hollow.

Fixed typos in 11.2, 12.6.5: changed “`JDODatastoreException`” to “`JDODataStoreException`”

Inserted new 11.4 to add `PersistenceManagerFactory` close method.

Added to 12.6 “In a non-managed environment, if the current transaction is active, `close()` throws `JDOUserException`.”

In 12.6.1, added new methods `retrieveAll (Collection, boolean)` and `retrieveAll (Object[], boolean)`.

In 12.6.1, clarified the description of retrieve.

In 12.6.4, clarified the description of getExtent to throw JDOUserException if the metadata does not require an extent to be maintained.

In 12.6.5, changed code example from `aPersistenceManager.getObjectById(pc.getPersistenceManager().getObjectId(pc), validate)` to `aPersistenceManager.getObjectById(JDOHelper.getObjectId(pc), validate)`. This avoids using the PersistenceCapable interface from user code.

In 12.6.5, changed the exception thrown by getObjectById to `JDOObjectNotFoundException`.

In 12.6.6, clarified description of makeTransient to make clear that the persistence manager is not responsible for clearing references to parameter instances to avoid making them persistent by reachability at commit.

In 12.6.6, clarified description of makeTransactional to include throwing JDOUnsupportedOptionException if a parameter is transient but TransientTransactional is not supported.

Fixed typo in 13.4.2. Changed “The `retainValues` setting currently active is returned.” to “The `restoreValues` setting currently active is returned.”

Fixed typo in 13.4.2. Changed “If this flag is set to `true`, then restoration of persistent instances does not take place after transaction rollback.” to “If this flag is set to `true`, then restoration of persistent instances takes place after transaction rollback.”

Corrected 13.4.3 to remove the requirement that Transaction must implement `javax.transaction.Synchronization`.

In 13.5, changed the behavior of failed optimistic transactions. The commit method throws a `JDOOptimisticVerificationException` and automatically rolls back the transaction.

Clarified 14.3 that variable declarations each require a type and a name, and there must be separating semicolons only if more than one declaration.

Clarified 14.3 that “candidate instances” are a subset of the candidate collection that are instances of the candidate class or a subset of the candidate class.

Clarified 14.4 that “compile time” refers to “JDOQL-compile time”.

Changed 14.5 to state “If the candidates are not specified, then the candidate extent is the extent of instances in the datastore with subclasses `true`.”

Clarified 14.6.2 if a cast operation would throw `ClassCastException`, it is treated the same as a `NullPointerException`.

Clarified 14.6.5 the semantics of “contains” is “exists”. This clarification is needed to provide a rational meaning if the contains clause is negated.

Clarified in 15 that Extents are not managed for instances of embedded fields.

In 15.3, clarified that the iterator method will throw an exception if `NontransactionalRead` is not supported.

In 17.1, added `getCause()`, `getFailedObject()` and `getNestedExceptions()` to the description of `JDOException`.

In 17.1, fixed description of `JDOUnsupportedOptionException`: “This class is a derived class of `JDOUserException`. This exception is thrown by an implementation to indicate that it does not implement a JDO optional feature.”

In 17.1.9, added new `JDOObjectNotFoundException` to report instances that cannot be found in the datastore.

In 17.1.10, added new `JDOOptimisticVerificationException` to report optimistic verification failures during commit.

Changed chapter 18 introduction to describe new policy for naming and accessing meta-data files.

In 18.3, changed name scoping for `persistence-capable-superclass`.

Corrected 18.4 to correct an inconsistency with 20.9.6: “null-valued fields throw a `JDOUserException` when the instance is flushed to the datastore and the datastore does not support null values.”

Clarified in 18.4 that Extents are not managed for instances of embedded fields.

Updated 18.4.1 and 18.4.2 to clarify type name scoping: The type names use Java rules for naming: if no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the “\$” marker.

In 18.6, added DOCTYPE description to describe access to the public DTD at java.sun.com/dtd.

Changed 19.3 to reflect change in portable object identity field types.

Changed 20.9.6 to correct an inconsistency with 18.4: “null-valued fields throw a `JDOUserException` when the instance is flushed to the datastore and the datastore does not support null values.”

Changed 20.17 and 20.20.4 to modify security checking for `JDOPermission` (“setStateManager”).

Changed 20.17 to correct the access modifier of `jdoPreSerialize` from private to protected.

Changed 20.20.1 to correct the interface name from `javax.jdo.PersistenceCapable` to `javax.jdo.spi.PersistenceCapable`.

Added new `JDOPermission` (“closePersistenceManagerFactory”) to check that the caller of `PersistenceManagerFactory.close()` is authorized.

Corrected Chapter 23 to remove alternative Name (`ArgumentListopt`) from `MethodInvocation` nonterminal in the BNF.

Corrected Chapter 23 to remove the exclusive or operator from the BNF.

Removed Appendix B.3 since it no longer reflects reality.

Index

A

accessDeclaredMembers 171
afterCompletion 41
application 34
associated object 95

B

beforeCompletion 41
begin 101
Binary Compatibility 19
Binary compatibility 138

C

Cache management 88
Change of identity 37
Cloning 143
Closing Query results 114
Collection 87
commit 101
compile 109
Connection 20, 24
connection 16, 23, 25, 96
Connection Management 97
ConnectionFactory 80
copyKeyFieldsToObjectId 75

D

declareImports 108
declareParameters 108
declareVariables 108
Delete persistent instances 93
deletePersistent 93, 123
Document Type Descriptor 133

E

ejbActivate 123
ejbCreate 123
ejbLoad 123
ejbPassivate 123
ejbRemove 123
ejbStore 123
ELEMENT array 132
ELEMENT class 129
ELEMENT collection 131
ELEMENT extension 132
ELEMENT field 130
ELEMENT jdo 129
ELEMENT map 132

ELEMENT package 129

evict 88
exceptions 125
execute 109
executeWithArray 110
executeWithMap 110
Extent 90, 119
Extent iterator 119

F

Field Numbering 142

G

Generated fields 148, 149
Generated methods 149
Generated static initializer 149
getFieldNames 72
getFieldTypes 72
getIgnoreCache 90, 108
getJDOImplHelper 171
getMultithreaded 94
getObjectById 90, 123
getObjectId 91, 123
getObjectIdClass 95
GetPersistenceManager 62
getPersistenceManager 81, 99, 107
getPersistenceManagerFactory 95
getSynchronization 101
getTransactionalObjectId 92
getUserObject 95

H

Hollow 42
home interface 123

I

IgnoreCache 108
Inheritance 60, 142
inheritance 129
Inner class 129
Instance life cycle management 92
InstanceCallbacks 76
Introspection (Java core reflection) 144
isActive 99

J

JDO Identity 35, 40, 56, 63, 68, 90, 113, 136, 167
JDO identity 38
JDO option 33, 34, 43

Index

jdoCopyFields 162
jdoCopyKeyFieldsToObjectId 150, 151, 163, 164
jdoFieldFlags 148
jdoFieldNames 149, 155
jdoFieldTypes 149, 155
jdoFlags 147, 154
jdoGetField 140, 141, 146, 158
jdoGetManagedFieldCount 157
jdoGetObjectId 63, 68, 69
jdoGetPersistenceManager 62
JDOHelper 62, 68
JDOImplHelper 72
jdoInheritedFieldCount 148, 154
jdoIsDeleted 64, 69
jdoIsDirty 63
jdoIsNew 64, 69
jdoIsPersistent 63, 69
jdoIsTransactional 63, 69
jdoMakeDirty 57, 62
jdoNewInstance 64, 151, 157
JDOPermission("getMetadata") 171
JDOPermission("setStateManager") 171
jdoPersistenceCapableSuperclass 149
jdoPostLoad 76
jdoPreClear 76
jdoPreSerialize 163
jdoPreStore 76
jdoProvideField 161
jdoProvideFields 161
jdoReplaceField 160
jdoReplaceFields 160
jdoReplaceStateManager 156, 157
jdoSetField 140, 141, 146, 159, 160, 161
jdoStateManager 154

M

Make instances nontransactional 94
Make instances persistent 92
Make instances transactional 93
Make instances transient 93
makeNontransactional 94
makePersistent 92, 123
makeTransactional 93
makeTransient 93

Membership 116
Multithreaded 94
N
Namespaces in queries 106
newInstance 73
newObjectIdInstance 75
newQuery 106
Nontransactional 43
NontransactionalRead 99
NullCollection 83, 113

O

Object Database 25
object database 18, 19, 104
object equality 35
object identity 35, 139
ObjectId class management 95
Optimistic 98, 99, 102
Optimistic transaction 46
Ordering 114

P

persistence by reachability 40
PersistenceCapable 62
PersistenceManager 85, 87
PersistenceManagerFactory 78
Persistent-clean 42
Persistent-deleted 42
Persistent-dirty 41
Persistent-new 40
Persistent-nontransactional 44
Portability Guidelines 135
primary key 35
Properties 82
provisionally persistent 92

Q

Query factory 89

R

ReflectPermission 171
refresh 88
registerClass 73, 155
relational 15, 18, 19, 25, 32, 104, 146
restoreValue 40
RestoreValues 43, 44, 70, 100
RetainValues 100
retrieve 89

Index

retrieveAll 89

rollback 101

S

Serialization 142

setCandidates 108

setClass 108

setEntityContext 123

setFilter 108

setIgnoreCache 89, 108

setMultithreaded 94

setNontransactionalRead 99

setNontransactionalWrite 99

setOptimistic 100

setOrdering 108

setRetainValues 100

setStateManager 171

setSynchronization 100

setUserObject 95

SQL 104

static initialization 149

static initializer 155

supported query languages 82

supportedOptions 82

suppressAccessChecks 171

Synchronization 94, 100

T

Threading 86

Transaction factory 89

Transient 40

Transient-clean 45

Transient-dirty 46

U

unsetEntityContext 123

V

validate 90

W

writeObject 163



4140 Network Circle
Santa Clara, CA 95404

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000