Java™ Data Objects 2.0

JSR 243

Final

22 February 2006

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

Specification: JSR-243,Java Data Objects - An Extension to the JDO specification ("Specification")

Version: 2.0

Status: Pre-FCS, Proposed Final Draft

Release: June 17, 2005

LIMITED EVALUATION LICENSE

DISCLAIMER OF WARRANTIES

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, RELATED IN ANY WAY TO YOUR HAVING OR USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

If you provide Sun with any comments or suggestions concerning the Specification ("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.

GOVERNING LAW

Any action relating to or arising out of this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

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.

Since the publication of JDO 1.0, many people have contributed time, energy, and ideas to the JDO effort. I want to recognize these significant contributors: Robin Roos, Abe White, David Jordan, Michael Bouschen, Michael Watzek, Wes Biggs, Geoff Hendry, Christian Romberg, David Tinker, Patrick Linskey, Bernhard Sporkmann, David Ezzio, Dion Almaer, Dirk Bartels, Dirk Theune, Eric Samson, Gordan Vosicki, Keiron McCammon, Matthew Adams, Oliver Kamps, Rod Johnson, Erik Bengtson, Andy Jefferson,Michelle Caisse, and Joerg von Frantzius.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Tables

List of Figures

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, aside from JDO, there are three Java standards for storing Java data persistently: serialization, JDBC, and Enterprise JavaBeans. 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. Enterprise JavaBeans require a container in which to run.

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 datastores 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.

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. 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

To use persistence-capable classes with binary-compatible JDO implementations, the classes must implement the `PersistenceCapable` contract, which includes implementing the `javax.jdo.spi.PersistenceCapable` contract, as well as adding other methods including static registration methods. This contract enables management of classes including transparent loading and storing of the fields of their persistent instances. A JDO enhancer, or byte code enhancer, is a program that modifies the byte codes of application-component Java class files to implement this interface.

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.

There are alternative approaches to byte code enhancement for having the classes implement the `PersistenceCapable` contract. These include preprocessing or code generation. If one of these alternatives is used instead of byte code enhancement, the `PersistenceCapable` contract is implemented explicitly.

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

A JDO implementation may optionally choose to support binary compatibility with other JDO implementations by supporting the `PersistenceCapable` contract for persistence-capable classes. If it does, then enhanced classes produced by another implementation or by the reference enhancer must be supported according to the following requirements.

- classes enhanced by the reference enhancer must be usable by any JDO compliant implementation that supports BinaryCompatibility;
- 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 that supports BinaryCompatibility.

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

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

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 participate 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, might update data in an underlying datastore, and do not directly represent data in the datastore.

Message-driven Beans

Message-driven beans are EJB application components that execute on behalf of a single client in response to an incoming message, might be transaction aware, might 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.

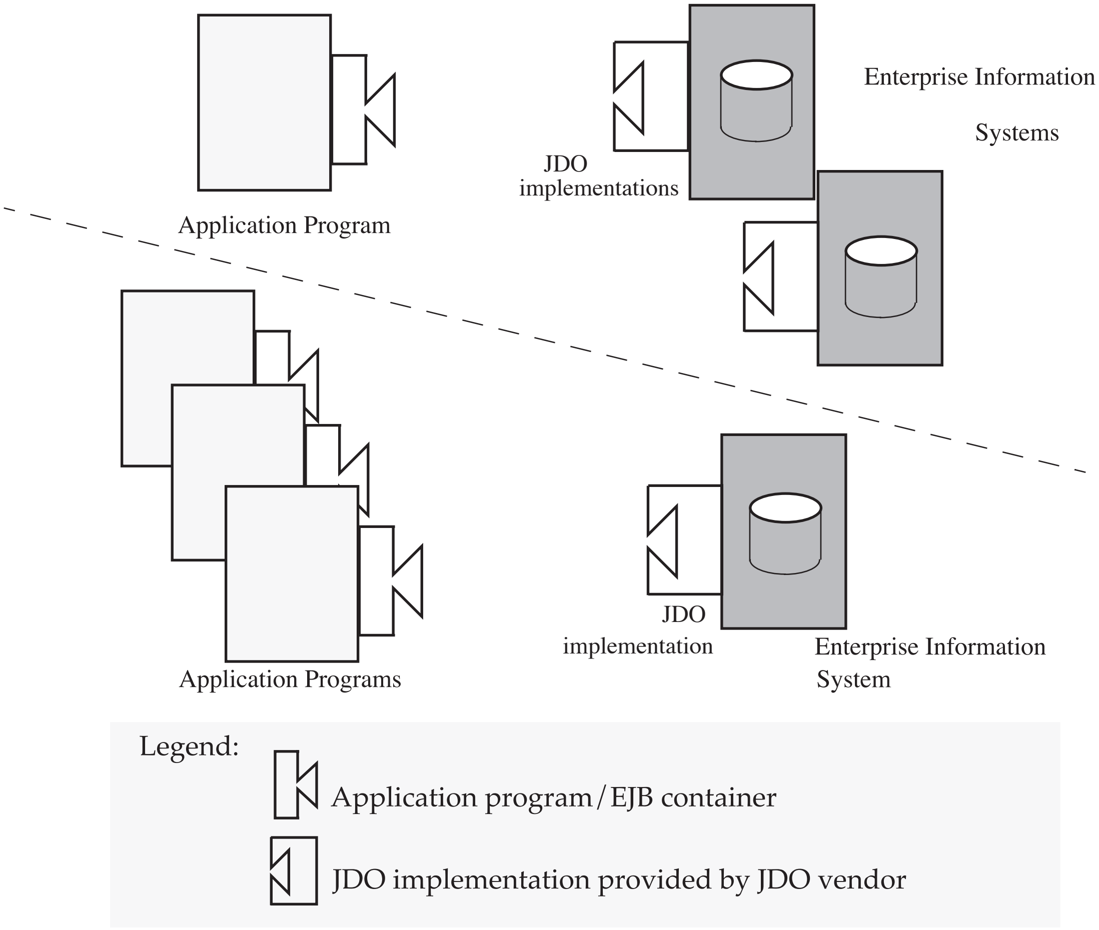
This architecture enables an EIS vendor to provide a standard data access interface for its EIS. The JDO implementation is plugged into an application server and provides underlying infrastructure for integration between the EIS and application components.

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 27 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.

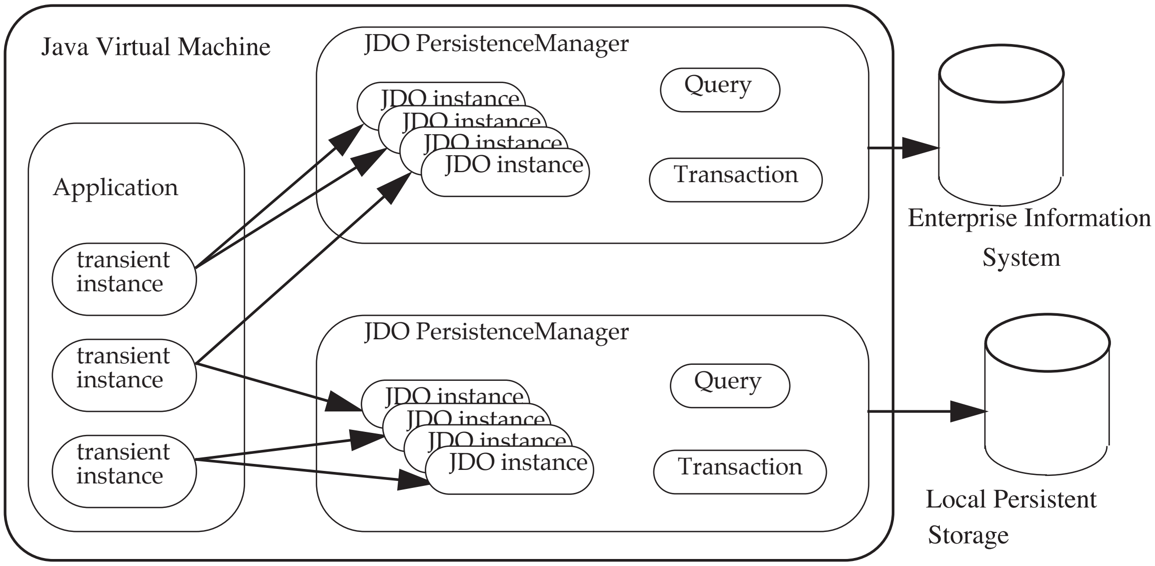
- The JDO architecture uses the J2EE Connector Architecture to make it applicable to all J2EE platform compliant application servers from multiple vendors.
- The JDO architecture makes it easy for application component developers to use the Java programming model to model the application domain and transparently 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.JDOHelper`, which provides the bootstrap and 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 `JDOHelper` class provides services such as bootstrap methods to acquire an instance of `PersistenceManagerFactory` and life cycle state interrogation 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 datastore. 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 `PersistenceManager`s 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 `PersistenceManager`s 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 `PersistenceManager`s.

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

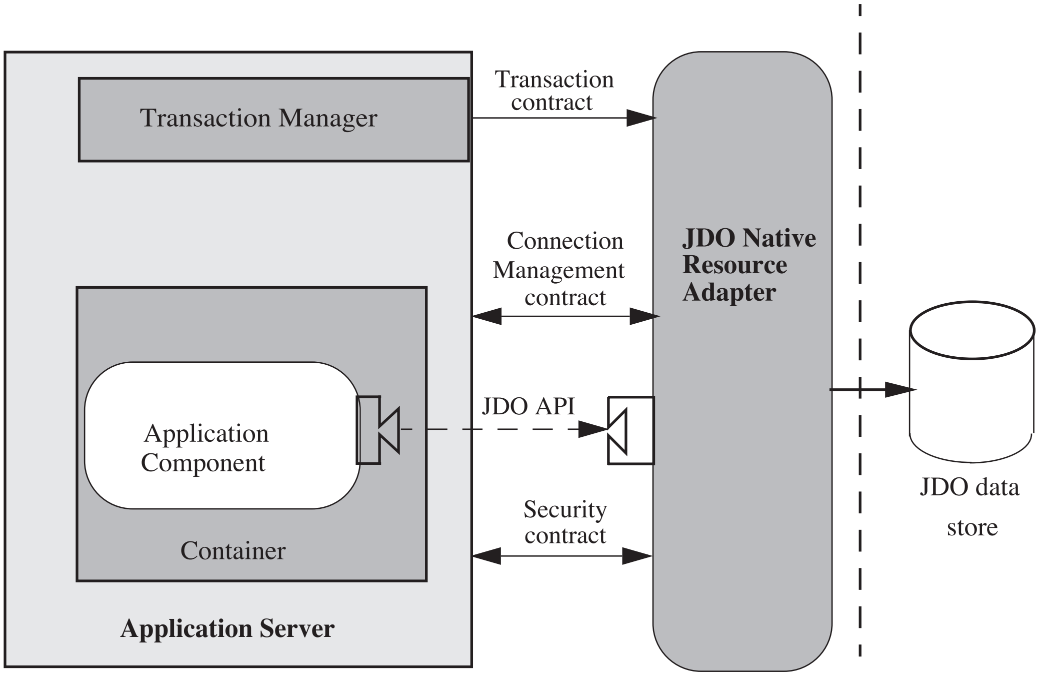
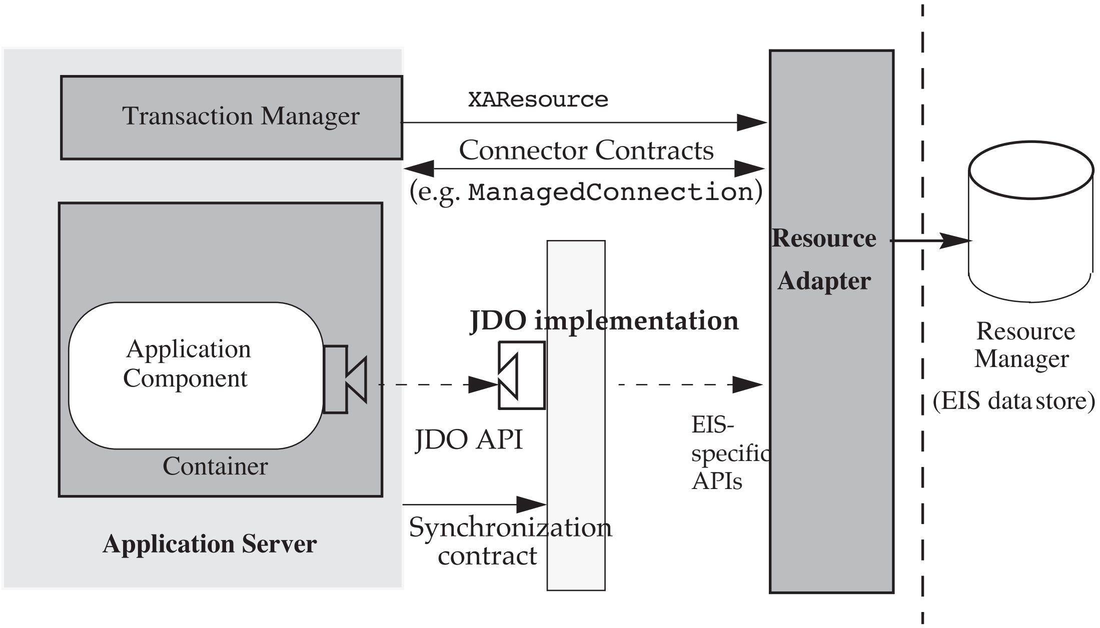


Figure 4.0 Contracts between application server and layered JDO implementation



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 (message-driven 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 `JDOHelper`. They can use the life cycle methods and query factory methods, while ignoring the transaction demarcation methods if they use container-managed transactions.

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], 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.

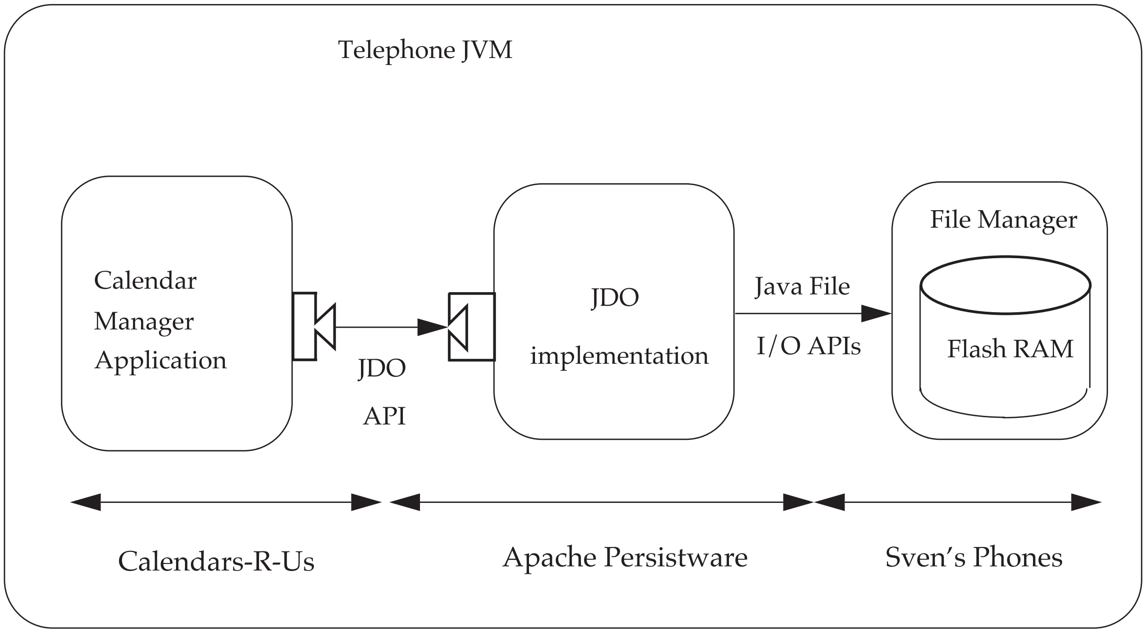
4.1.9 Container Provider

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.

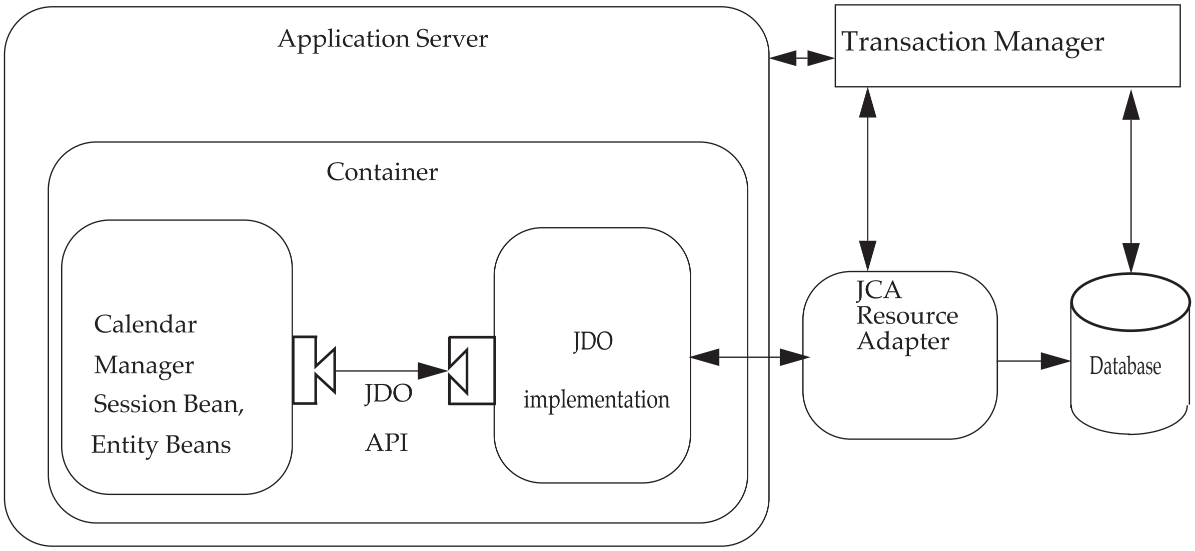
Using the JDO interface, Calendars-R-Us rewrote just the query part of their software. The application classes did not have to be changed. Only the persistence interface that queried for specific instances needed to be modified.

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 for binary compatible implementations, 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 transient, detached, 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 the instance behaves exactly like an ordinary instance of the 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 (outside the JDO environment) modifications to the datastore.

If a JDO instance is detached, its behavior is very similar to that of a transient instance, with a few significant exceptions. A detached instance does not necessarily have all of its persistent fields loaded from the data store, and any attempt to access unloaded fields, whether for read or write, is denied. A detached instance maintains its persistent identity and the identity can be obtained by an observer. A detached instance allows changes to be made to loaded fields, and tracks those changes while detached. Detached instances never observe transaction boundaries.

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. Detached instances might be 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.

In a binary-compatible implementation, 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 `PersistenceManager`s 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. In a binary-compatible implementation, 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. In a binary-compatible implementation, 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. For both binary-compatible and non-binary-compatible implementations, these contract extensions support interrogation of the life cycle state of the instances and are intended for use by management parts of the system.

Persistent instances might be constructed by the application and made persistent; or might be constructed by the JDO `PersistenceManager` in response to a query or navigation from a persistent instance or via the `newInstance` method. If the JDO `PersistenceManager` constructs the instance, the class of the instance might be a derived class of the class of the original instance, and will respond true to `instanceof` the class of the original. Thus, applications must not rely on tests of the actual class of persistent instances, but must instead use the `instanceof` test.

JDO State Manager

In a binary-compatible implementation, 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. The Java VM implements object identity via the = = operator. This can be used by JDO implementations to determine whether two instances are identical (have the same location) in the VM.

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 application implements `hashCode` and `equals`, to create the application's vision of equality of instances, typically based on values of fields in the instances. The JDO implementation must not use the application's `hashCode` and `equals` methods from the persistence-capable classes except as needed to implement the Collections Framework in package `java.util`. The JDO implementation must use the application's `hashCode` and `equals` methods from the application-provided object id classes.

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 `PersistenceManager`s 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. The JDO implementation is responsible for the implementation of JDO identity based on the user's declaration of the identity type of each persistence-capable class.

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 persistence-capable class and is fixed at class loading 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 `JDOUnsupportedOptionException` being thrown. The exception might be thrown immediately or upon flush or transaction commit.

> *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. The exception might be thrown immediately or upon flush or transaction commit.

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 constructor taking either a `String` alone or a `Class` and `String` 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 derived from the least-derived (topmost) concrete persistence-capable class in the hierarchy.

Compound Identity

Compound identity is a special case of application identity. References to other persistence-capable classes can be defined as key fields. In this case, the object id class contains a field that is of the type of the object id of the relationship field.

For example, two classes have a one-many relationship, and on the reference side of the relationship, the field is a key field. On the other side of the relationship, there is a `Collection` or other multi-valued type.

```
class Order {
long orderId;
Set<OrderItem> items;
...}
class OrderId {
long orderId; // matches orderId field name
...}
class OrderItem {
Order order;
long item;
...}
class OrderItemId {
OrderId order; // matches order field name
long item; matches item field name
...}
```

5.4.2 Single Field Identity

A common case of application identity uses exactly one persistent field in the class to represent identity. In this case, the application can use a standard JDO class instead of creating a new user-defined class for the purpose.

A JDO implementation that supports application identity must also support single field identity.

```
package javax.jdo.identity;
public abstract class SingleFieldIdentity implements Externalizable
{
     protected SingleFieldIdentity(Class pcClass);
     public Class getTargetClass();
     public String getTargetClassName();
     public Object getKeyAsObject();
}

public class ByteIdentity
     extends SingleFieldIdentity {
          public byte getKey();
          public ByteIdentity(Class pcClass, byte key);
          public ByteIdentity(Class pcClass, Byte key);
          public ByteIdentity(Class pcClass, String key);
     }

     public class CharIdentity
          extends SingleFieldIdentity {
          public char getKey();
          public CharIdentity(Class pcClass, char key);
          public CharIdentity(Class pcClass, Character key);
          public CharIdentity(Class pcClass, String key);
     }
     public class ShortIdentity
          extends SingleFieldIdentity {
          public short getKey();
          public ShortIdentity(Class pcClass, short key);
          public ShortIdentity(Class pcClass, Short key);
          public ShortIdentity(Class pcClass, String key);
     }

     public class IntIdentity
          extends SingleFieldIdentity {
          public int getKey();
          public IntIdentity(Class pcClass, int key);
          public IntIdentity(Class pcClass, Integer key);
          public IntIdentity(Class pcClass, String key);
     }

     public class LongIdentity
          extends SingleFieldIdentity {
          public long getKey();
          public LongIdentity(Class pcClass, long key);
          public LongIdentity(Class pcClass, Long key);
          public LongIdentity(Class pcClass, String key);
     }

     public class StringIdentity
          extends SingleFieldIdentity {
          public String getKey();
          public StringIdentity(Class pcClass, String key);
     }

     public class ObjectIdentity
          extends SingleFieldIdentity {
          public Object getKey();
          public ObjectIdentity(Class pcClass, Object key);
     }
```

The constructors that take reference types throw `JDONullIdentityException` if the second argument is null. Valid key values are never null.

Constructors of primitive identity types that take String parameters convert the parameter to the proper type using the static parseXXX method of the corresponding wrapper class.

Instances of `SingleFieldIdentity` classes are immutable. When serialized, the name of the target class is serialized. When deserialized, the name of the target class is restored, but not the target class. The deserialized instance will return `null` to `getTargetClass`. All instances will return the "binary" name of the target class (the result of `Class.getName()`).

The `SingleFieldIdentity` classes adhere to all of the requirements for application object id classes, with the exception of field names. That is, there are no public fields visible to the application.

5.4.3 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 (e.g. `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 the `PersistenceManager` method `newObjectIdInstance(Class cls, String key)`;

Note that, unlike application 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.

5.4.4 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 many states defined by this specification. Some states are required, and others states are optional. If an implementation does not support certain operations, then these optional 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 or compiler-generated 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 one of the `PersistenceManager` `makePersistent` methods 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 `before-`

`Completion`, 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.If `restoreValues` is `true`, the values of managed fields in the instance are restored to the values as they were at the time makePersistent 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 `StoreCallback`.

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.

A hollow instance transitions to detached if the transaction associated with its persistence manager is committed while the `DetachAllOnCommit` property is true.

The behavior of persistent instances at close of the corresponding `PersistenceManager` is not further defined in this specification.

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. Any other access to persistent fields is not supported and might 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. Any other access to persistent fields is not supported and might 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.

5.5.8 Detached-clean (Required)

JDO instances that have been detached from their persistence manager and have not been modified are detached-clean. Detach is done by one of three ways:

- the instance or an instance containing a reference to the instance is serialized; in this case, the serialized instance is detached
- the transaction of the persistence manager managing the instance is committed and the `DetachAllOnCommit` property is `true`; in this case the persistent instance itself is detached (there is no copy)
- the instance is explicitly detached from the persistence manager via one of the `detachCopy` or `detachCopyAll` methods; in this case the copy is detached.

Detached-clean instances transition to detached-dirty if a loaded field is modified. Attempts to change their state via any of the persistence manager methods except for `makePersistent` and `deletePersistent`throw `JDOUserException`.

`Evict`, `refresh`, `retrieve`, `makeTransient`, `makeTransactional`, `makeNon-transactional`, and `detachCopy` throw `JDOUserException` if a parameter instance is in the detached-clean or detached-dirty state.

A detachable class is not serialization-compatible with the corresponding unenhanced class.

Detached instances are further described in section 12.6.8.

5.5.9 Detached-dirty (Required)

JDO instances that have been removed from their persistence manager and have fields marked as modified are detached-dirty.

Fields are marked as modified if a field of the detached instance is explicitly modified by the application.

Detached-dirty instances do not change their life cycle state.

`Evict`, `refresh`, `retrieve`, `makeTransient`, `makeTransactional`, `makeNon-transactional`, and `detachCopy` throw `JDOUserException` if a parameter instance is in the detached-clean or detached-dirty state.

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 might participate in a subsequent transaction.
- This is an incompatible change from the behavior in JDO 1.0. Compatibility is only supported if a subsequent transaction is not begun after making changes to persistent instances in the cache.

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. These changes might be committed in a subsequent transaction.

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.

A persistent-nontransactional instance transitions to detached if a transaction is commited while the `DetachAllOnCommit` property is `true`.

A persistent-nontransactional instance transitions to persistent-nontransactional-dirty if a change is made outside a transaction while the `NontransactionalWrite` property is `true`.

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.

The behavior of persistent instances at close of the corresponding `PersistenceManager` is not further defined in this specification.

5.6.2 Persistent-nontransactional-dirty (Optional)

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

The persistent-nontransactional-dirty state allows applications to operate on nontransactional instances in the cache and make changes to the instances without having a transaction active. At some future point, the application can begin a transaction and incorporate the changes into the transactional state. Committing the transaction makes the changes made outside the transaction durable.

A persistent-nontransactional-dirty instance transitions to hollow if it is the parameter of `evict` or `evictAll`. This allows the application to remove instances from the set of instances whose state is to be committed to the datastore.

If a datastore transaction is begun, `commit` will write the changes to the datastore with no checking as to the current state of the instances in the datastore. That is, the changes made outside the transaction together with any changes made inside the transaction will overwrite the current state of the datastore. The persistent-nontransactional-dirty instances will transition according to the `RetainValues` flag. With the `RetainValues` flag set to `true`, persistent-nontransactional-dirty instances will transition to persistent-nontransactional. With the `RetainValues` flag set to `false`, persistent-nontransactional-dirty instances will transition to hollow.

If a datastore transaction is begun, `rollback` will not write any changes to the datastore. The persistent-nontransactional-dirty instances will transition according to the `RestoreValues` flag. With the `RestoreValues` flag set to `true`, persistent-nontransactional-dirty instances will make no state transition, but the fields will be restored to their values as of the beginning of the transaction, and any changes made within the transaction will be discarded. With the `RestoreValues` flag set to `false`, persistent-nontransactional-dirty instances will transition to hollow.

If an optimistic transaction is begun, `commit` will write the changes to the datastore after checking as to the current state of the instances in the datastore. The changes made outside the transaction together with any changes made inside the transaction will update the current state of the datastore if the version checking is successful. The persistent-nontransactional-dirty instances will transition according to the `RetainValues` flag. With the `RetainValues` flag set to `true`, persistent-nontransactional-dirty instances will transition to persistent-nontransactional. With the `RetainValues` flag set to `false`, persistent-nontransactional-dirty instances will transition to hollow.

If an optimistic transaction is begun, `rollback` will not write any changes to the datastore. The persistent-nontransactional-dirty instances will transition according to the `RestoreValues` flag. With the `RestoreValues` flag set to `true`, persistent-nontransactional-dirty instances will make no state transition, but the fields will be restored to their values as of the beginning of the transaction, and any changes made within the transaction will be discarded. With the `RestoreValues` flag set to `false`, persistent-nontransactional-dirty instances will transition to hollow.

The behavior of persistent instances at close of the corresponding `PersistenceManager` is not further defined in this specification.

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 the optional feature `javax.jdo.option.TransientTransactional`.

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. This behavior is not dependent on the setting of the `RestoreValues` flag.

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

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 the optional feature `javax.jdo.option.TransientTransactional`.

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. This behavior is not dependent on the setting of the `RestoreValues` flag.

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 Optimistic transaction	unchanged	unchanged	unchanged	P-nontrans	unchanged
evict	n/a	unchanged	Hollow	unchanged	unchanged
read field outside transaction	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
commit transaction with DetachAllOnCommit true	unchanged	detached-clean	detached-clean	detached-clean	detached-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 Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
commit transaction with DetachAllOnCommit true	unchanged	unchanged	Transient	Transient	detached-clean

method \ current state	P-nontrans-dirty	detached-clean	detached-dirty
makePersistent	unchanged	unchanged	unchanged
deletePersistent	error	unchanged	unchanged
makeTransactional	unchanged	error	error
makeNontransactional	error	error	error
makeTransient	error	error	error
commit with retainValues=false	hollow	unchanged	unchanged
commit with retainValues=true	P-nontrans	unchanged	unchanged
rollback	unchanged	unchanged	unchanged
refresh	unchanged	error	error
evict	hollow	error	error
read field	unchanged	unchanged	unchanged
write field or makeDirty	unchanged	detached-dirty	unchanged
retrieve	unchanged	error	error
commit transaction with DetachAllOnCommit true	detached	unchanged	unchanged

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

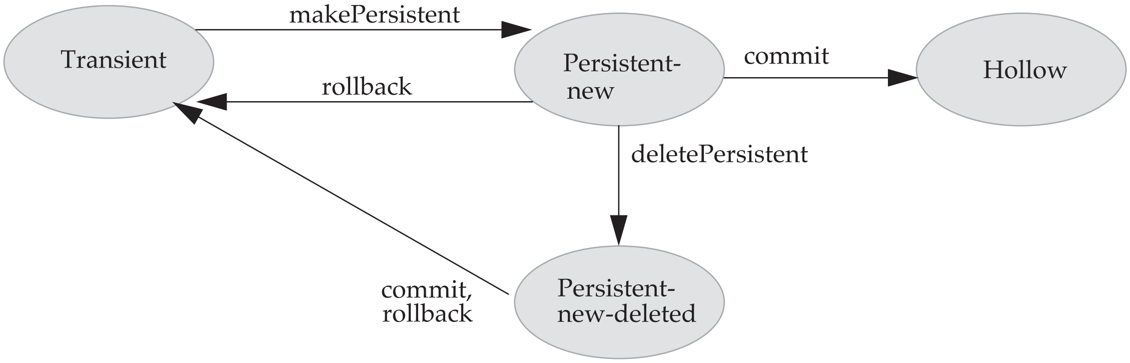


Figure 8.0 Life Cycle: Transactional Access

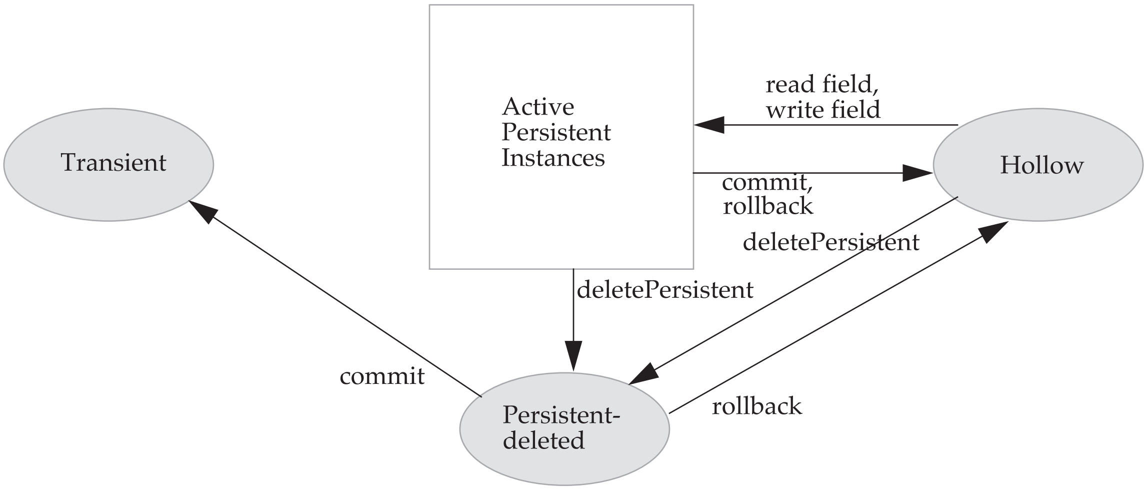


Figure 9.0 Life Cycle: Datastore Transactions

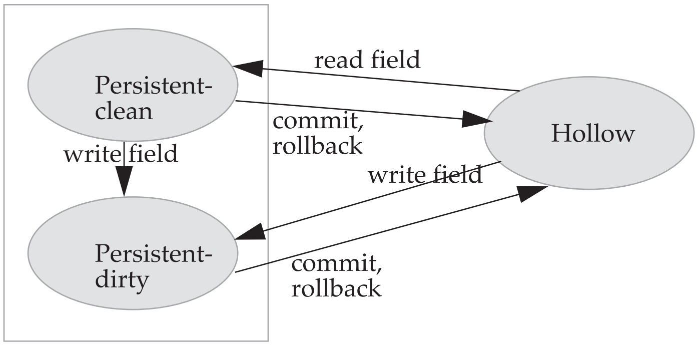


Figure 10.0 Life Cycle: Optimistic Transactions

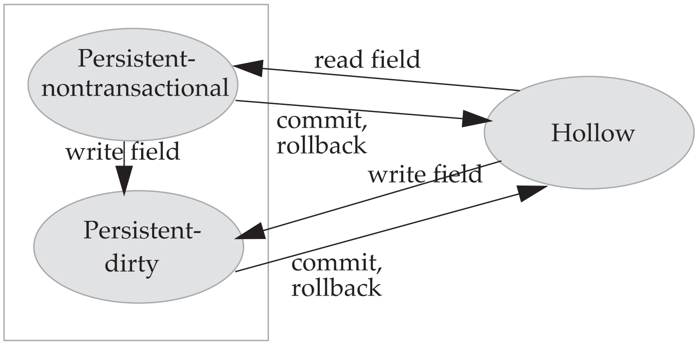


Figure 11.0 Life Cycle: Access Outside Transactions

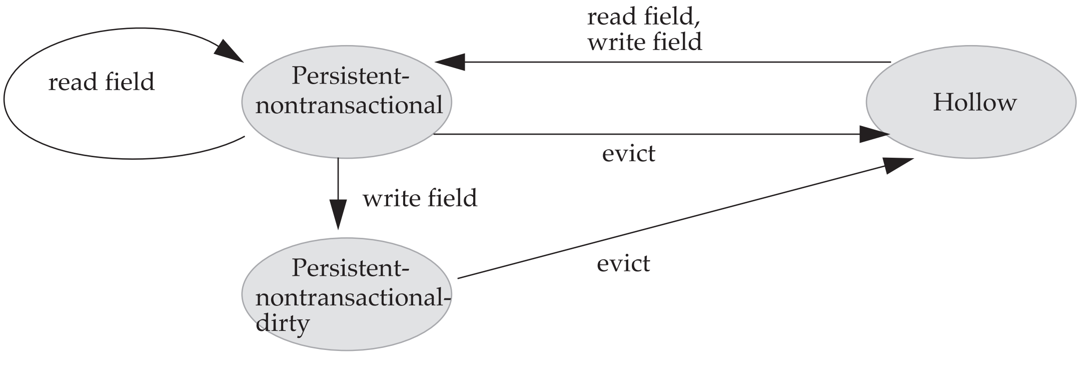


Figure 12.0 Life Cycle: Transient TransactionalLife Cycle: Transient Transactional

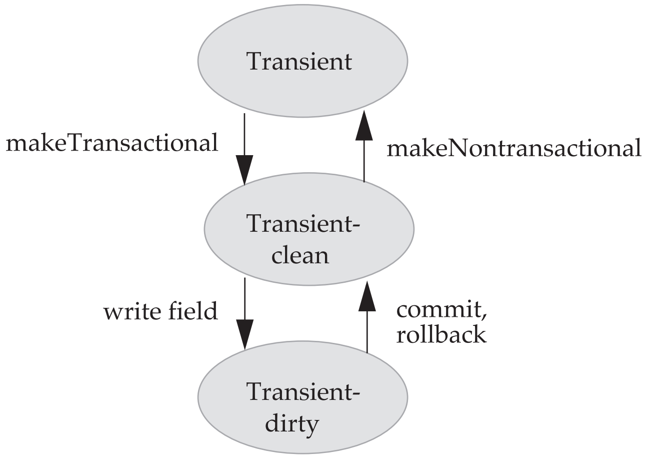


Figure 13.0 Life Cycle: Detached

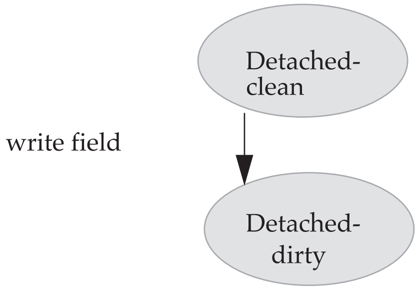
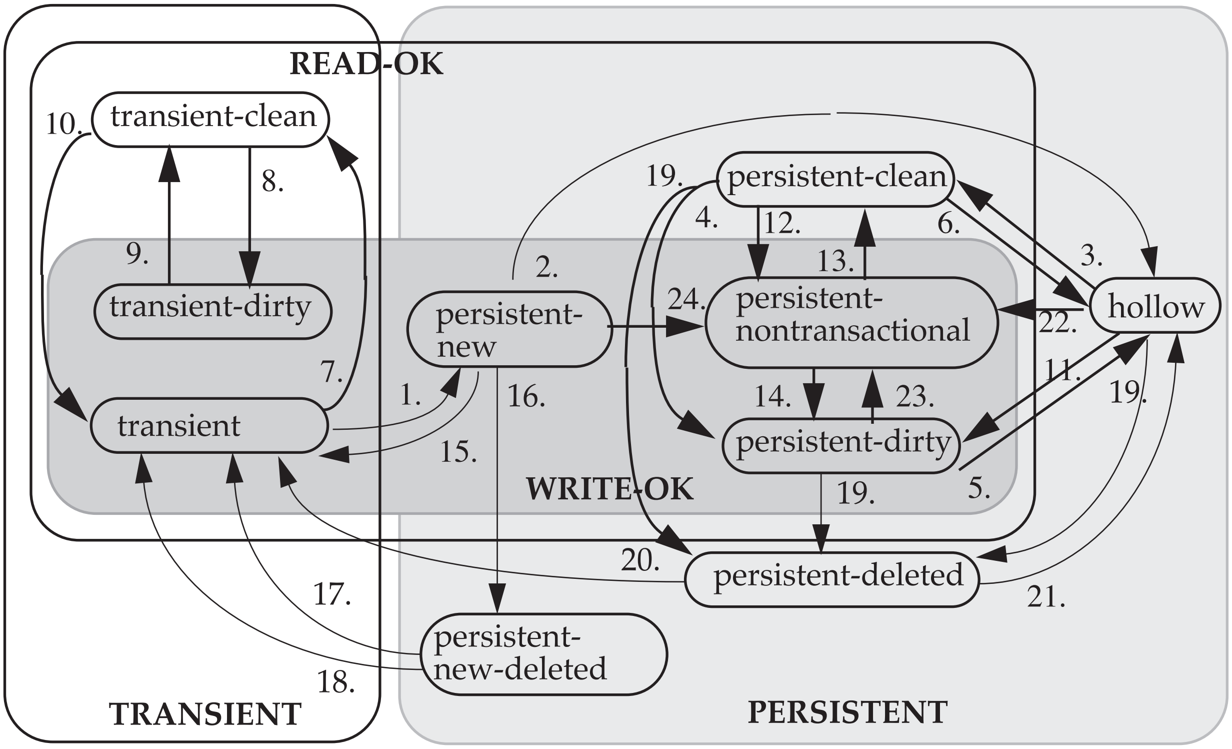


Figure 14.0 JDO Instance State Transitions



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

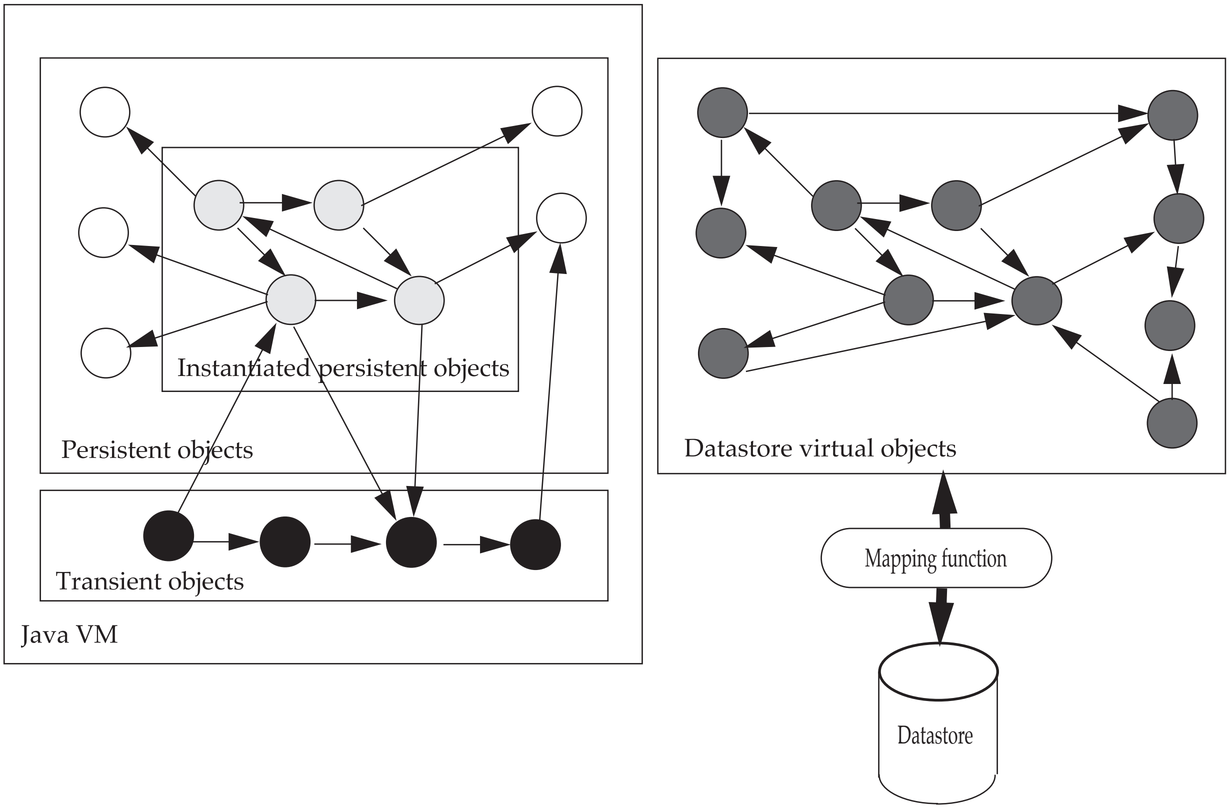
This chapter specifies the object model for persistence capable classes. To the extent possible, the object model is the same as the Java object model. Differences between the Java object model and the JDO object model are highlighted.

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 15.0 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 except in the case of relational databases, for which a subset of mapping functionality is standard. The standard part of the mapping is specified in Chapter 15.

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.

Persistence-capable

The JDO Object Model distinguishes between two kinds of classes: those that are marked as persistence-capable and those that aren't. A user-defined class can be persistence-capable 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 `java.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 a persistence-capable class that has a JDO Identity, can be stored in a datastore, and can be 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 instances of a persistence-capable 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 persistence-capable 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 persistence-capable type, it is not visible to queries of the `Extent` of the persistence-capable class. Furthermore, the field values of SCO instances of persistence-capable 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 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.

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 `makeDirty` method 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 class `JDOHelper` 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

Precision of fields

JDO implementations may not represent Java types precisely in the datastore, because not all datastores are able to natively represent all Java types. Some type mapping may be required. The precision of the mapping is a quality of service issue with the JDO implementation and the particular datastore.

The mapping precision restriction applies to the range of values that can be faithfully stored and retrieved, the precision of the values, and the scale of BigDecimal values.

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`, `Currency.`
- package `java.math`: `BigDecimal`, `BigInteger`.

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

The scale of `BigDecimal` values is not guaranteed to be preserved by implementations. For example, saving a persistent field with value `BigDecimal("1.2300")` might be returned as value `BigDecimal("1.23")`.

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`, `HashMap, Hashtable, LinkedHashMap, LinkedHashSet`.

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`, `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.

Persistence-capable Class types

JDO implementations must support references to FCO instances of persistence-capable classes and are permitted, but not required, to support references to SCO instances of persistence-capable classes.

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, Map,` and `Set` are required; `List` is 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 some classes in the inheritance hierarchy to be persistence-capable and some not persistence-capable.

The expression "`obj instanceof PersistenceCapable`" can be true (because of a persistence-capable superclass) when in fact the class of obj is not persistence-capable. Thus, it is not possible for an application to examine a class to determine whether an instance of that class is allowed to be persistent.

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.

Readers primarily interested in developing applications with the JDO API can ignore the following chapter. Skip to 8 – JDOHelper.

7 PersistenceCapable

For JDO implementations that support the BinaryCompatibility rules, 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`. Other enhancers can be used for specific binary-compatible JDO implementations.

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 (and only portable) approach for applications to interrogate the state of persistence-capable instances is to use the class `JDOHelper`, which provides static methods that delegate to the instance if it implements `PersistenceCapable`, and if not, attempts to find the JDO implementation responsible for the instance, and if unable to do so, returns the values that would have been returned by a transient instance.

Classes that are to be detached from the persistence manager further implement the `Detachable` interface. This interface is used to establish the fields loaded before detachment and to query the instance if it is presented for attachment later.

The persistence modifier, identity type, identity class, key fields, persistent fields, and detachability of the class are fixed at enhancement time, or when the class is loaded, whichever occurs first.

> *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;
public 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);void jdoMakeDirty (int fieldNumber);`

These methods mark the specified field dirty so that its values will be modified in the datastore 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 instance 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. The `fieldNumber` parameter is the internal field number assigned during class enhancement.

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. If the instance is detached, this method returns the identity as of the time of detachment.

`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. If the instance is detached, this method returns the identity as of the time of detachment.

7.3.1 Version

`Object jdoGetVersion();`

This method returns the version of the instance.

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 or detached, `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 or detached, `false` is returned.

7.4.3 Persistent

`boolean jdoIsPersistent();`

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

7.4.6 Detached

`boolean jdoIsDetached();`

Table 3: State interrogation

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

Table 3: State interrogation

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

Instances that have been detached return `true`.

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("setStateManager")` 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);
void jdoCopyField (Object other, int fieldNumber);
```

These methods copy fields from another instance of the same class. These methods 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.

For classes using single field identity, this method must be called on an instance of a persistence-capable class with its primary key field initialized (not null), or a `JDONullIdentityException` is thrown.

The instance returned is initialized with the value(s) of the primary key field(s) of the instance on which the method is called.

`public Object jdoNewObjectIdInstance(Object key);`

This method creates a new instance of the class used for JDO identity, using the appropriate constructor of the object id class. It is intended only for application identity, including single field identity. If the class has been enhanced for datastore identity, or if the class is abstract, `null` is returned. The identity instance returned has no relationship with the values of the primary key fields of the persistence-capable instance on which the method is called.

For single field identity, there is specific behavior required for parameters of these types:

- `ObjectIdFieldSupplier`: the field value is fetched and used to construct the single field identity instance.
- `Number` or `Character`: the parameter `key` must be an instance of the key type or, for primitive key types, the wrapper of the key type; the key is passed as a parameter to the single field identity constructor.
- `String`: the `String` is parsed to a value of the appropriate type and the value is used to construct the single field identity instance. For `ObjectIdentity`, the `String` is decomposed into two parts using ":" as a delimiter. The first part is the class name; the second is the `String` representation of the value of the class.
- `Object`: for `ObjectIdentity`, the key type must be assignable from the parameter `key`.

`public void jdoCopyKeyFieldsToObjectId(Object oid);`

This method copies all key fields from this instance to the parameter. The parameter must be an instance of the JDO identity class, or `ClassCastException` is thrown. If the class uses single field identity, this method always throws `JDOFatalInternalException`.

`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. If the class uses single field identity, this method always throws `JDOFatalInternalException`.

`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

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

7.13 Detachable

This interface contains the method used by the `StateManager` to manage the detached state in a detached instance. This interface is not intended to be used by application programs.

The detached state is stored as a field in each instance of `Detachable`. The field is serialized so as to maintain the state of the instance while detached. While detached, only the `BitSet` of modified fields will be modified. The structure is as follows.

```
Object[] jdoDetachedState;
```

```
jdoDetachedState[0]: the Object Id of the instance
jdoDetachedState[1]: the Version of the instance
jdoDetachedState[2]: a BitSet of loaded fields
jdoDetachedState[3]: a BitSet of modified fields
package javax.jdo.spi;
public interface Detachable {
void jdoReplaceDetachedState();
}
```

This method calls the `StateManager` with the current detached state instance as a parameter and replaces the current detached state instance with the value provided by the `StateManager`.

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, it delegates to any JDO implementations registered with `JDOImplHelper` via the `StateInterrogation` interface.

If no registered implementation recognizes the instance, then

- if the method returns a value of reference type, it returns `null`;
- if the method returns a value of boolean type, it returns `false`;

if the method returns `void`, there is no effect.If no registered implementation recognizes the instance, then

- if the method returns a value of reference type, it returns `null`;
- if the method returns a value of boolean type, it returns `false`;

if the method returns `void`, there is no effect.If no registered implementation recognizes the instance, then

- if the method returns a value of reference type, it returns `null`;
- if the method returns a value of boolean type, it returns `false`;
- 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 if its class is not persistence-capable.

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 datastore 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 if its class is not persistence-capable; 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 for persistent and detached instances. It returns `null` if the instance is transient or `null` or if its class is not persistence-capable. 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[] getObjectIds (Object[] pcs);`

`static Collection getObjectIds (Collection pcs);`

These methods return the JDO identities of the parameter instances. For each instance in the parameter, the `getObjectId` method is called. They return one identity instance for each persistence-capable instance in the parameter. The order of iteration of the returned `Collection` exactly matches the order of iteration of the parameter `Collection`.

`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 JDO Version

`static Object getVersion (Object pc);`

This method returns the JDO version of the instance for persistent and detached instances. It returns `null` if the instance is transient or `null` or if its class is not persistence-capable.

8.5 Status interrogation

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

8.5.1 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 if its class is not persistence-capable.

See also `PersistenceCapable.jdoIsDirty();`

8.5.2 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 if its class is not persistence-capable.

See also `PersistenceCapable.jdoIsTransactional()`.

8.5.3 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 if its class is not persistence-capable.

See also `PersistenceCapable.jdoIsPersistent();`

8.5.4 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 if its class is not persistence-capable.

See also `PersistenceCapable.jdoIsNew();`

8.5.5 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 if its class is not persistence-capable.

See also `PersistenceCapable.jdoIsDeleted();`

8.5.6 Detached

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

Instances that have been detached return true. The method returns false if the instance is transient or null or if its class is not detachable.

See also `PersistenceCapable.jdoIsDetached();`

8.6 PersistenceManagerFactory methods

```
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (Map props, ClassLoader loader);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (Map props);
```

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

```
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (File propsFile);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (File propsFile, ClassLoader loader);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
```

```
      (String propsResourceName);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (String propsResourceName, ClassLoader loader);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (String propsResourceName, ClassLoader propsLoader,
         ClassLoader pmfLoader);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (InputStream stream);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (InputStream stream, ClassLoader loader);public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (String jndiLocation, Context context);
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (String jndiLocation, Context context, ClassLoader loader);
```

These methods use the parameter(s) passed as arguments to construct a `Properties` instance, and then delegate to the static method `getPersistenceManagerFactory` in the class named in the property `javax.jdo.PersistenceManagerFactoryClass`. If there are any exceptions while trying to construct the `Properties` instance or 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.

The method taking a `String` as the `propsResourceName` argument uses the `propsLoader` to load the properties and uses the `pmfLoader` to resolve the `PersistenceManagerFactory` class name. The method taking a `String` as the `propsResourceName` argument with one `ClassLoader` uses the parameter `ClassLoader` to load both the properties and the `PersistenceManagerFactory` class name.The method taking a `String` alone uses the context class loader for both purposes.

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(Map)` method, then `JDOFatalInternalException` is thrown. If the implementation of the static `getPersistenceManagerFactory(Map)` 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
javax.jdo.option.Mapping
```

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.

```
public static
   PersistenceManagerFactory getPersistenceManagerFactory
      (String jndiName, Context context);
```

This method looks up the `PersistenceManagerFactory` using the naming context and name supplied. The implementation’s factory method is not called. The behavior of this method depends on the implementation of the context and its interaction with the saved `PersistenceManagerFactory` object. As with the other factory methods, the returned `PersistenceManagerFactory` is not configurable.

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 class 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

```
public 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 `ObjectId` 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, Object key);`

This method creates a new instance of the `ObjectId` class for the `PersistenceCapable` class, using the appropriate 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 Object newObjectIdInstance(PersistenceCapable pc);`

This method returns an instance of the `ObjectId` class for the parameter `PersistenceCapable` instance. If the class of the instance uses an immutable `ObjectId` class, then the oid instance associated with the persistent instance might be returned. If the class of the instance uses datastore identity, then `null` is returned.

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

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

9.7 Persistence-capable class state interrogation

For JDO implementations that do not support BinaryCompatibility, an instance of `StateInterrogation` must be registered with `JDOImplHelper` to handle `JDOHelper` methods for instances that do not implement `PersistenceCapable`.

The `StateInterrogation` interface is implemented by a JDO implementation class to take responsibility for determining the life cycle state and object identity, and for marking fields dirty.

```
package javax.jdo.spi;
public interface StateInterrogation {
Boolean isPersistent(Object pc);
Boolean isTransactional(Object pc);
Boolean isDirty(Object pc);
Boolean isNew(Object pc);
Boolean isDeleted(Object pc);
Boolean isDetached(Object pc);
PersistenceManager getPersistenceManager(Object pc);
Object getObjectId(Object pc);
Object getTransactionalObjectId(Object pc);
boolean makeDirty(Object pc, String fieldName);
Object getVersion(Object pc);
}
```

For methods returning `Boolean`, `PersistenceManager`, and `Object`, if the `StateInterrogation` instance does not recognize the parameter instance, `null` is returned, and the next registered `StateInterrogation` instance is called.

For `makeDirty`, if the `StateInterrogation` instance does not recognize the parameter instance, `false` is returned, and the next registered `StateInterrogation` instance is called.

```
public void addStateInterrogation(StateInterrogation si);
```

This method of `JDOImplHelper` registers an instance of `StateInterrogation` for delegation of life cycle state queries made on JDOHelper.

```
public void removeStateInterrogation(StateInterrogation si);
```

This method of `JDOImplHelper` removes an instance of `StateInterrogation`, so it is no longer called by `JDOHelper` for life cycle state queries.

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 the class implements the corresponding callback interface . For backward compatibility, `InstanceCallbacks` is redefined as follows:

```
package javax.jdo;
public interface InstanceCallbacks extends
   javax.jdo.listener.LoadCallback,
   javax.jdo.listener.StoreCallback,
   javax.jdo.listener.ClearCallback,
   javax.jdo.listener.DeleteCallback {
}
```

10.1 jdoPostLoad

```
package javax.jdo.listener;
public interface LoadCallback {
void jdoPostLoad();
}
```

This method is called after values have been loaded from the `StateManager` into the instance, if an active fetch group has been defined with the `post-load` attribute set to `true`. Non-persistent fields whose value depends on values of loaded fields should be initialized in this method. This method is not modified by the enhancer. Only fields that are loaded by an active 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

```
package javax.jdo.listener;
public interface StoreCallback {
```

```
void jdoPreStore();
}
```

This method is called before the values are stored from the instance to the datastore. This happens during `beforeCompletion` and `flush` for persistent-new and persistent-dirty instances of persistence-capable classes that implement `StoreCallback`. 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

```
package javax.jdo.listener;
public interface ClearCallback {
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

```
package javax.jdo.listener;
public interface DeleteCallback {
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.

10.5 jdoPreDetach and jdoPostDetach

```
package javax.jdo.listener;
public interface DetachCallback {
void jdoPreDetach();
```

This method is called during the execution of `detachCopy` on the persistent instance before the copy is made.

`public void jdoPostDetach(Object detached);`

This method is called during the execution of `detachCopy` on the detached instance after the copy is made. The parameter is the corresponding persistent instance.

`}`

10.6 jdoPreAttach and jdoPostAttach

`package javax.jdo.listener;`

`public interface AttachCallback {`

`void jdoPreAttach();`

This method is called during the execution of `makePersistent` on the detached instance before the copy is made.

`public void jdoPostAttach(Object attached);`

This method is called during the execution of `makePersistent` on the persistent instance after the copy is made. The parameter is the corresponding detached instance.

`}`

11 PersistenceManagerFactory

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

```
package javax.jdo;
public 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.
- `DetachAllOnCommit`: the `PersistenceManager` mode that indicates that instances will be detached when the transaction commits.

The following properties can only be set in the `PersistenceManagerFactory`:

`Mapping`: the name of the mapping model for object-to-datastore mapping`Catalog`: the name of the catalog for object-to-relational mapping`Schema`: the name of the schema for object-to-relational mapping

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 `Map` instance. This static method is called by the `JDOHelper` method `getPersistenceManagerFactory (Map props)`.

```
static PersistenceManagerFactory getPersistenceManagerFactory
(Map 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.DetachAllOnCommit"`
- `"javax.jdo.option.ConnectionUserName"`
- `"javax.jdo.option.ConnectionPassword"`
- `"javax.jdo.option.ConnectionURL"`
- `"javax.jdo.option.ConnectionDriverName"`
- `"javax.jdo.option.ConnectionFactoryName"`
- `"javax.jdo.option.ConnectionFactory2Name"`
- `"javax.jdo.option.Mapping"`
- `"javax.jdo.mapping.Catalog"`
- `"javax.jdo.mapping.Schema"`

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`.

The `Mapping` property specifies the object-data store mapping to be used by the implementation. The property is used to construct the names of resource files containing metadata. For more information on the use of this property, see Chapters 15 and 18.

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

There are properties that are provided by the `JDOHelper` methods in the following cases.

- If the user uses the methods `getPersistenceManagerFactory(File file)` or `getPersistenceManagerFactory(File file, ClassLoader loader)` then the `Map` instance passed to the static method will contain a property with a key of `“javax.jdo.spi.PropertiesFileName”`, and a value equal to the result of calling `getAbsolutePath()` on the file parameter. Absence of this property means that neither of these methods was used.
- If the user uses the methods `getPersistenceManagerFactory(String resourceName)` or `getPersistenceManagerFactory(String resourceName, ClassLoader loader)` then the `Properties` instance passed to the static method will contain a property with a key of `”javax.jdo.spi.PropertiesResourceName”`, and a value equal to the name of the resource. Absence of this property means that neither of these methods was used.

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 these 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.This method will never return the same instance as was returned by a previous invocation of the method. Note that this implies that pooled implementations must use proxies and not return the identical pooled 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` and mark this `PersistenceManagerFactory` as closed. After `close` completes, disallow all methods except `close`, `isClosed`, and `get` methods except for `getPersistenceManager`. If any disallowed method is called after `close`, then `JDOUserException` is thrown.`boolean isClosed();`

Return `true` if this `PersistenceManagerFactory` is closed; and `false` otherwise.

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`

The JDO implementation supports the transient transactional life cycle states.

`javax.jdo.option.NontransactionalRead`

The JDO implementation supports reading and querying outside a transaction.

`javax.jdo.option.NontransactionalWrite`

The JDO implementation supports the persistent-nontransactional-dirty life cycle state.

`javax.jdo.option.RetainValues`

The JDO implementation supports retaining values of persistent instances after commit.

`javax.jdo.option.Optimistic`

The JDO implementation supports the optimistic transaction semantics.

`javax.jdo.option.ApplicationIdentity`

The JDO implementation supports application identity for persistent classes.

`javax.jdo.option.DatastoreIdentity`

The JDO implementation supports datastore identity for persistent classes.

`javax.jdo.option.NonDurableIdentity`

The JDO implementation supports nondurable identity for persistent classes `javax.jdo.option.ArrayList`

The JDO implementation supports persistent field types of `ArrayList`.

`javax.jdo.option.LinkedList`

The JDO implementation supports persistent field types of `LinkedList`.

`javax.jdo.option.TreeMap`

The JDO implementation supports persistent field types of `TreeMap`.

`javax.jdo.option.TreeSet`

The JDO implementation supports persistent field types of `TreeSet`.

`javax.jdo.option.Vector`

The JDO implementation supports persistent field types of `Vector`.

`javax.jdo.option.List`

The JDO implementation supports persistent field types of `List`. This is now a requirement but the option is for compatibility with JDO 1.0 where this support was optional.

`javax.jdo.option.Array`

The JDO implementation supports persistent field types of array.

`javax.jdo.option.NullCollection`

The JDO implementation allows null collections to be stored. Most relational implementations do not distinguish between empty and null collections, and this option will not be set for those implementations.

`javax.jdo.option.ChangeApplicationIdentity`

The JDO implementation supports changing of the application identity of instances.

`javax.jdo.option.BinaryCompatibility`

The JDO implementation supports the binary compatibility contract.

`javax.jdo.option.GetDataStoreConnection`

The JDO implementation supports use of a direct datastore connection.

`javax.jdo.option.GetJDBCConnection`

The JDO implementation supports use of a direct datastore connection that implements the `java.sql.Connection` interface.

`javax.jdo.query.SQL`

The JDO implementation supports SQL for queries executed via the `javax.jdo.Query` interface.

`javax.jdo.option.UnconstrainedQueryVariables`

The JDO implementation supports JDOQL queries that contain a variable without a contains clause to constrain the variable.

`javax.jdo.option.version.DateTime`

The JDO implementation supports use of a the date-time strategy for version checking.

`javax.jdo.option.version.StateImage`

The JDO implementation supports use of the state-image strategy for version checking.

`javax.jdo.option.PreDirtyEvent`

The JDO implementation supports event notifications of changes made to persistent instances before the instance is made dirty.

`javax.jdo.option.mapping.HeterogeneousObjectType`

The JDO implementation supports mapping a persistent field of type Object to multiple types. There is no standard way to map this support.

`javax.jdo.option.mapping.HeterogeneousInterfaceType`

The JDO implementation supports mapping a persistent field of a persistent interface type to multiple types. There is no standard way to map this support.

`javax.jdo.option.mapping.JoinedTablePerClass`

The JDO implementation supports mapping persistent class inheritance hierarchies to tables in which each class, including abstract classes, is mapped to a table; and each table mapped to a subclass defines a primary key that has a foreign key relationship to the primary key of the table mapped by the superclass.

`javax.jdo.option.mapping.JoinedTablePerConcreteClass`

The JDO implementation supports mapping persistent class inheritance hierarchies to tables in which each concrete class (excluding abstract classes) is mapped to a table; and each table mapped to a subclass defines a primary key that has a foreign key relationship to the primary key of the table mapped by the superclass.

`javax.jdo.option.mapping.NonJoinedTablePerConcreteClass`

The JDO implementation supports mapping persistent class inheritance hierarchies to tables in which each concrete class (excluding abstract classes) is mapped to a table; and there is not necessarily any foreign key relationship among the mapped tables.

`javax.jdo.option.mapping.RelationSubclassTable`

The JDO implementation supports mapping persistent fields containing relationships to classes in an inheritance relationship that use subclass-table as the field mapping strategy.

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 (Map props);
```

This static method is not a method defined in the `PersistenceManagerFactory` interface, but rather must be defined on the class that implements `PersistenceManagerFactory`. It returns an instance of `PersistenceManagerFactory` based on the properties in the parameter.

The method is used by `JDOHelper` to construct an instance of `PersistenceManagerFactory` based on user-specified properties.

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

`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`

`javax.jdo.option.Mapping`

`javax.jdo.mapping.Catalog`

`javax.jdo.mapping.Schema`

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.

11.8 Second-level cache management

Most JDO implementations allow instances to be cached in a second-level cache, and allow direct management of the cache by knowledgeable applications. The second-level cache is typically a single VM cache and is used for persistent instances associated with a single `PersistenceManagerFactory`. For the purpose of standardizing this behavior, the `DataStoreCache` interface is used.

To obtain a reference to the cache manager, the `getDataStoreCache()` method of `PersistenceManagerFactory` is used.

```
DataStoreCache getDataStoreCache();
```

If there is no second-level cache, the returned instance silently does nothing.

```
package javax.jdo.datastore;
public interface DataStoreCache {
```

Evicting objects from the cache

```
   void evict(Object oid);
   void evictAll();
   void evictAll(Object[] oids);
   void evictAll(Collection oids);
   void evictAll(Class pcClass, boolean subclasses);
```

The evict methods are hints to the implementation that the instances referred to by the object ids are stale and should be evicted from the cache. Evicting an instance does not unpin it.

Pinning objects in the cache

```
void pin(Object oid);
void pinAll(Collection oids);
void pinAll(Object[] oids);
void pinAll(Class pcClass, boolean subclasses);
```

The pin methods are hints to the implementation that the instances referred to by the object ids should be pinned in the cache (not subject to algorithm-based eviction, but subject to explicit eviction). There is no requirement that an instance be in the cache in order to pin or unpin it. The `pinAll` method with the `Class` parameter automatically pins all instances of that class, including those instances already in the cache and future instances of the class. When a class is pinned, pin and unpin methods on instances of the pinned class are ignored.

Unpinning objects in the cache

```
void unpin(Object oid);
void unpinAll(Collection oids);
void unpinAll(Object[] oids);
void unpinAll(Class pcClass, boolean subclasses);
```

The unpin methods are hints to the implementation that the instances referred to by the object ids should be unpinned (subject to eviction based on algorithm). There is no requirement that an instance be in the cache in order to pin or unpin it. The `unpinAll` method with the `Class` parameter automatically unpins all instances of that class, including those instances already in the cache and future instances of the class. When a class is pinned, pin and unpin methods on instances of the pinned class are ignored.

}

11.9 Registering for life cycle events

```
void addInstanceLifecycleListener (InstanceLifecycleListener listener, Class[] classes);
```

This `PersistenceManagerFactory` method adds the listener to the list of instance lifecycle event listeners set as the initial listeners for each `PersistenceManager` created by this `PersistenceManagerFactory`. The classes parameter identifies all of the classes of interest. If the classes parameter is specified as `null`, events for all persistent classes and interfaces are generated. If the classes specified have persistence-capable subclasses, all such subclasses are registered implicitly.

The listener will be called for each event for which it implements the corresponding listener interface.

```
void removeInstanceLifecycleListener (InstanceLifecycleListener listener);
```

This `PersistenceManagerFactory` method removes the listener from the list of event listeners set as the initial listeners for each `PersistenceManager` created by this `PersistenceManagerFactory`.

The `addInstanceLifecycleListener` and `removeInstanceLifecycleListener` methods are considered to be configuration methods and can only be called when the `PersistenceManagerFactory` is configurable (before the first `getPersistenceManager` is called).

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 persistence-capable 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 persistence-capable classes 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, boolean initialize, ClassLoader loader)` with up to three loaders. The `initialize` parameter can be either `true` or `false` depending on the implementation.

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, or the loader of the object passed to the `newQuery` method.
   - 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 the application calls `PersistenceManagerFactory.getPersistenceManager()`. This loader is saved with the `PersistenceManager` and cleared when the `PersistenceManager` is closed.

12.6 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 or detached state. As soon as a transient instance is made persistent or transactional, it will be associated with exactly one JDO `PersistenceManager`.Detached instances are never associated with a `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, for implementations that support BinaryCompatibility, JDO instances passed as parameters and returned as values must implement `PersistenceCapable`. The interface defines these formal parameters as `Object` because binary compatibility is optional.

```
package javax.jdo;
public interface 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(), close(),` and `get` methods throw a `JDOFatalUserException`.

State Transitions for persistent instances at close

The behavior of persistent instances at close of the corresponding `PersistenceManager` is not further defined in this specification.

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 contain `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 transactional 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`.

If the parameter instance is detached, then `JDOUserException` is thrown.

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);
void refreshAll (JDOException ex);
```

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.

When called with a transaction active, 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.

If there is a fetch plan in effect, then the fetch plan affects the results of this method. All modified fields and all fields in the current fetch plan are unloaded and then fields in the current fetch plan are fetched from the datastore.

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

When used with the `JDOException` parameter, the JDO `PersistenceManager` refreshes all instances in the exception, including instances in nested exceptions, that failed verification. Updated and unchanged instances that failed verification are reloaded from the datastore. Datastore instances corresponding to new instances that failed due to duplicate key are loaded from the datastore.

If the parameter instance is detached, then `JDOUserException` is thrown.

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

These methods request the `PersistenceManager` to load 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 the current fetch plan or a vendor-specific pre-read policy (not specified by JDO).

If the `FGOnly` parameter is `false`, or the method without the `FGOnly` parameter is invoked, all fields must be loaded from the datastore.

If the `FGOnly` parameter is `true`, and the fetch plan has not been modified from its default setting (see 12.7.5), then this is a hint to the implementation that only the fields in the current 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 `FGOnly` parameter `true`, all fields in the current fetch group must have been fetched, but other fields might be fetched lazily by the implementation.

If the `FGOnly` parameter is `true`, and the fetch plan has been changed, then only the fields specified by the fetch plan are loaded.

If the parameter instance or instances are detached, then `JDOUserException` is thrown.

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;
- if the class of the instance implements `LoadCallback` calls `jdoPostLoad`;
- calls `postLoad` for all `LifecycleListener` instances that are registered for load callbacks for the class of the loaded instances.

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 persistence capable classes and interfaces. If an instance of a class or interface that has a managed extent is made persistent via reachability, the instance is put into the extent implicitly. If an instance of a class that implements an interface that has a managed extent is made persistent, then that instance is put into the interface's extent.

`Extent getExtent (Class persistenceCapable, boolean subclasses);`

`Extent getExtent (Class persistenceCapable);`

The `getExtent` method returns an `Extent` that contains all of the instances in the parameter class or interface, and if the subclasses flag is `true`, all of the instances of the parameter class and its subclasses. The method with no subclasses parameter is treated as equivalent to `getExtent (persistenceCapable, true)`.

If the metadata does not indicate via the `requires-extent` attribute in the `class` or `interface` element that an extent is managed for the parameter class or interface, 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.

Extents of interfaces

If the `Class` parameter of the `getExtent` method is an interface, then the interface must be identified in the metadata as having its extent managed.

12.6.5 JDO Identity management

`Object getObjectById (Object oid);`

The `getObjectById` method attempts to find an instance in the cache with the specified JDO identity. This method behaves exactly as the method `getObjectById (Object oid, boolean validate)` with the `validate` flag set to `true`.

`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 persistence-capable 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 the fetch plan has been changed from its original value, the fetch plan governs which fields are fetched from the datastore and which related objects are also fetched with them.
  - 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.

Getting Multiple Persistent Instances

```
Collection getObjectsById (Collection oids);
Object[] getObjectsById (Object[] oids);
Collection getObjectsById (Collection oids, boolean validate);
Object[] getObjectsById (Object[] oids, boolean validate);
```

The `getObjectsById` method attempts to find instances in the cache with the specified JDO identities. The elements of the `oids` parameter object might have been returned by earlier calls to `getObjectId` or `getTransactionalObjectId`, or might have been constructed by the application.

If a method with no `validate` parameter is used, the method behaves exactly as the corresponding method with the `validate` flag set to `true`.

If the `Object[]` form of the method is used, the returned objects correspond by position with the object ids in the oids parameter. If the `Collection` form of the method is used, the iterator over the returned `Collection` returns instances in the same order as the oids returned by an iterator over the parameter `Collection`. The cardinality of the return value is the same as the cardinality of the `oids` parameter.

Getting an Object by Class and Key

```
Object getObjectById (Class cls, Object key);
```

The `getObjectById` method attempts to find an instance in the cache with the derived JDO identity. The `key` parameter is either the string representation of the object id, or is an object representation of a single field identity key.

This is a convenience method that exactly matches the behavior of calling `pm.getObjectById (pm.newObjectIdInstance (cls, key), true)`.

12.6.6 Persistent instance factory

The following method is used to create an instance of a persistence-capable interface, or of a concrete or abstract class.

```
Object newInstance(Class persistenceCapable);
```

The parameter must be one of the following:

- an abstract class that is declared in the metadata using the `class` element, or
- an interface that is declared in the metadata using the `interface` element, or
- a concrete class that is declared in the metadata as persistence-capable. In this case, the concrete class must declare a public no-args constructor.

The returned instance is transient, and is an "`instanceof`" the parameter. Applications might use the instance via the `get` and `set` property methods and change its life cycle state exactly as if it were an instance of a persistence-capable class.

In order for the `newInstance` method to be used, the parameter interface must be completely mapped. For relational implementations, the interface must be mapped to a table and all persistent properties must be mapped to columns. Additionally, interfaces that are the targets of all relationships from persistent properties must also be mapped. Otherwise, `JDOUserException` is thrown by the `newInstance` method.

For interfaces and classes that use a `SingleFieldIdentity` as the object-id class, if the returned instance is subsequently made persistent, the target class stored in the object-id instance is the parameter of the `newInstance` method that created it.

12.6.7 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

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

These methods make transient instances persistent and apply detached instance changes to the cache. They must be called in the context of an active transaction, or a `JDOUserException` is thrown. For a transient instance, 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.For a detached instance, they locate or create a persistent instance with the same JDO identity as the detached instance, and merge the persistent state of the detached instance into the persistent instance. Only the state of persistent fields is merged. If non-persistent state needs to be copied, the application should use the `jdoPostAttach` callback or the `postAttach` lifecycle event listener. Any references to the detached instances from instances in the closure of the parameter instances are modified to refer to the corresponding persistent instance instead of to the detached instance.

During application of changes of the detached state, if the JDO implementation can determine that there were no changes made during detachment, then the implementation is not required to mark the corresponding instance dirty. If it cannot determine if changes were made, then it must mark the instance dirty.

No consistency checking is done during `makePersistent` of detached instances. If consistency checking is required by the application, then `flush` or `checkConsistency` should be called after attaching the instances.

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.The return value for instances in the transient or persistent states is the same as the parameter value. The return value for detached instances is the persistent instance corresponding to the detached instance.

The return values for `makePersistentAll` methods correspond by position to the parameter instances.

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. However, if a field is marked as containing a dependent reference, the dependent instance is deleted as well.

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.

If the parameter instance of `deletePersistent()` is a detached instance, the method applies to the associated persistent instance. Similarly, if any of the parameter instances of `deletePersistentAll()` is a detached instance, the method applies to the associated persistent instances. If the class of any instance to be deleted implements `DeleteCallback`, or if there are any `InstanceLifecycleListeners` registered for deletion callbacks of instances of any detached objects' class, then the parameter persistent instances of those classes are instantiated, the callback is executed and/or the listeners are called with the event, as described in section 12.15.

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 these methods 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.

If the parameter instance or instances are detached, then `JDOUserException` is thrown.

These methods will be ignored if the instance is transient.

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

If the `useFetchPlan` parameter is `false`, these methods behave exactly as the corresponding `makeTransient` methods.

If the `useFetchPlan` parameter is `true`, the current `FetchPlan`, including `MaxFetchDepth, DETACH_LOAD_FIELDS`, and `DETACH_UNLOAD_FIELDS`, is applied to the `pc` or `pcs` parameter instance(s) to load fields and instances from the datastore. The `DetachmentRoots` is not affected. After the fetch plan is used to load instances, the entire graph of instances reachable via loaded fields of the parameter instances is made transient. Transient fields are not modified by the method.

If the parameter instance or instances are detached, then `JDOUserException` is thrown.

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 `JDOUnsupportedOptionException` is thrown.

If the parameter instance or instances are detached, then `JDOUserException` 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 the parameter instance or instances are detached, then `JDOUserException` is thrown.

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.

12.6.8 Detaching and attaching instances

These methods provide a way for an application to identify persistent instances, obtain copies of these persistent instances, modify the detached instances either in the same JVM or in a different JVM, apply the changes to the same or different `PersistenceManager`, and commit the changes.

There are three ways to cause the creation of detached instances:

- explicitly via methods defined on `PersistenceManager`;
- implicitly by committing the transaction while the `DetachAllOnCommit` flag is true;
- or implicitly by serializing persistent instances.

Committing the transaction with DetachAllOnCommit

```
boolean getDetachAllOnCommit();
```

The value of the `DetachAllOnCommit` flag is returned.

```
void setDetachAllOnCommit(boolean flag);
```

The value of the `DetachAllOnCommit` flag is set to the parameter value. The flag takes effect during the next commit after being called. This method is allowed at any time except during transaction completion (`beforeCompletion` and `afterCompletion`).

In JDO 1.0, the behavior of persistent instances after closing the associated `PersistenceManager` is undefined. JDO 2 defines a new property called `DetachAllOnCommit` which changes this behavior. With this flag set to `false`, the state of persistent instances in the cache after commit is defined by the `retainValues` flag.

With this flag set to `true`, during `beforeCompletion` all cached instances are prepared for detachment according to the fetch plan in effect at commit. Loading fields and unloading fields required by the fetch plan is done after calling the user's `beforeCompletion` callback. During `afterCompletion`, before calling the user's `afterCompletion` callback, all detachable persistent instances in the cache transition to detached; non-detachable persistent instances transition to transient; and detachable instances can be serialized as detached instances. Transient transactional instances are unaffected by this flag.

Serializing Persistent Instances

The JDO 1.0 specification requires that serialized instances be made ready for serialization by instantiating all serializable persistent fields before calling `writeObject`. For binary-compatible implementations, this is done by the enhancer adding a call to the `StateManager` prior to invoking the user's `writeObject` method. The behavior is the same in JDO 2.0, with the additional requirement that restored detachable serialized instances are treated as detached instances.

Explicit detach

```
Object detachCopy(Object pc);
```

```
Collection detachCopyAll(Collection pcs);
```

```
Object[] detachCopyAll(Object[] pcs);
```

This method makes detached copies of the parameter instances and returns the copies as the result of the method. The order of instances in the parameter `Collection`'s iteration corresponds to the order of corresponding instances in the returned `Collection`'s iteration.Only persistent fields are copied by the JDO implementation. If transient fields need to be copied, the application should implement the `jdoPreDetach` callback or the `preDetach` lifecycle event listener.

If a `detachCopy` method is called with an active transaction, the parameter `Collection` of instances is first made persistent, and the reachability algorithm is run on the instances. This ensures that the closure of all of the instances in the the parameter `Collection` is persistent.

If a `detachCopy` method is called outside an active transaction, the reachability algorithm will not be run; if any transient instances are reachable via persistent fields, a `JDOUserException` is thrown for each persistent instance containing such fields.

If the parameter instance is detached, then `JDOUserException` is thrown.

If a `detachCopy` method is called outside an active transaction, the `NontransactionalRead` property must be true or `JDOUserException` is thrown.

For each instance in the parameter `Collection`, a corresponding detached copy is returned. Each field in the persistent instance is handled based on its type and whether the field is contained in the fetch group for the persistence-capable class. If there are duplicates in the parameter `Collection`, the corresponding detached copy is used for each such duplicate.

Instances in the persistent-new and persistent-dirty state are updated with their current object identity and version (as if they had been flushed to the datastore prior to copying their state). This ensures that the object identity and version (if any) is properly set prior to creating the copy. The transaction in which the flush is performed is assumed to commit; if the transaction rolls back, then the detached instances become invalid (they no longer refer to the correct version of the datastore instances). This situation will be detected at the subsequent attempt to commit or flush a transaction after attaching the detached instances.

If instances in a deleted state (either persistent-deleted or persistent-new-deleted) are attempted to be detached, a `JDOUserException` is thrown with nested `JDOUserExceptions`, one for each such instance.

Instances to be detached that are not of a `Detachable` class are detached as transient instances.

The `FetchPlan` in effect in the `PersistenceManager` at the time of detachment determines the fields to be fetched in the closure of the persistent instances. If the default fetch plan is active, instances are detached in their current state. If the user has changed the fetch plan, then each instance to be detached will have the fetch plan applied to it, including detachment options. The DETACH_LOAD_FIELDS causes the fields in the fetch plan to be loaded before the instances are detached. The DETACH_UNLOAD_FIELDS causes loaded fields that are not in the fetch plan to be unloaded before detachment.

Fields in the `FetchPlan` of primitive and wrapper types are set to their values from the datastore. Fields of references to persistence-capable types are set to the detached copy corresponding to the persistent instance. Fields of `Collections` and `Maps` are set to detached SCO instances containing references to detached copies corresponding to persistent instances in the datastore.

The result of the `detachCopyAll` method is a `Collection` or array of detached instances whose closure contains copies of detached instances. Among the closure of detached instances there are no references to persistent instances; all such references from the persistent instances have been replaced by the corresponding detached instance.

There might or might not be a transaction active when the `detachCopy` method is called.

Behavior of Detached Instances

While detached, any field access to a field that was not loaded throws `JDODetachedFieldAccessException`.

While detached, each detached instance has a persistent identity that can be obtained via the static `JDOHelper` method `getObjectId(Object pc)`. The version of detached instances can be obtained by the static `JDOHelper` method `getVersion(Object pc)`.

While detached, identity fields of application-identity classes might be modified by the application. These fields are marked as modified by the detached instance, but the object id of the detached instance does not change. Upon attachment, the change will be rejected if the jdo implementation does not support application identity change. See `PersistenceManagerFactory` property `javax.jdo.option.ChangeApplicationIdentity.`

Changes made to embedded instances of mutable types including persistence-capable types are tracked by the detached instance if they are replaced or modified. Changes are reflected by marking the detached instance's field as modified.

To apply changes made to instances while detached, use the `makePersistent` method with the detached instance as parameter.

12.7 Fetch Plan

A fetch plan defines rules for instantiating the loaded state for an object graph. It specifies fields to be loaded for all of the instances in the graph. Using fetch plans, users can control the field fetching behavior of many JDO APIs. A fetch plan can be associated with a `PersistenceManager` and, independently, with a `Query` and with an `Extent`.

A fetch plan also defines rules for creating the detached object graph for the detach APIs and for automatic detachment at `commit` with `DetachAllOnCommit` set to `true`.

A fetch plan consists of a number of fetch groups that are combined additively for each affected class; a fetch size that governs the number of instances of multi-valued fields retrieved by queries; a recursion-depth per field that governs the recursion depth of the object graph fetched for that field; a maximum fetch depth that governs the depth of the object graph fetched starting with the root objects; and flags that govern the behavior of detachment.

The default fetch plan contains exactly one fetch group, "`default`". It has a fetch size of 0, and detachment option DETACH_LOAD_FIELDS. The default fetch plan is in effect when the `PersistenceManager` is first acquired from the `PersistenceManagerFactory`.

With the default fetch plan in effect, the behavior of JDO 2 is very similar to the behavior of JDO 1. That is, when instances are loaded into memory in response to queries or navigation, fields in the default fetch group are loaded, and the `jdoPostLoad` callback is executed the first time an instance is fetched from the datastore. The implementation is allowed to load additional fields, as in JDO 1. Upon detachment, fields that are have been loaded into the detached instances are preserved, regardless of whether they were loaded automatically by the implementation or loaded in response to application access; and fields that have not been loaded are marked in the detached instances as not loaded.

This behavior is sufficient for the most basic use case for detachment, where the detached instances are simply “data transfer objects” containing primitive fields. The detached instances can be modified in place or serialized and sent to another tier to be changed and sent back. Upon being received back, the instances can be attached and if there are no version conflicts, the changes can be applied to the datastore.

The most common use case for fetch groups is to restrict the fields loaded for an instance to the primitive values and avoid loading related instances for queries. For more control over the default behavior, the “default” fetch group can simply be redefined for specific classes. For example, a `String` field that contains a typically large document can be defined as not part of the default fetch group, and the field will be loaded only when accessed by the application. Similarly, an `Order` field associated with `OrderLine` might be defined as part of the default fetch group of `OrderLine`, and queries on `OrderLine` will always load the corresponding `Order` instance as well. This can easily improve the performance of applications that always need the `Order` whenever `OrderLine` instances are loaded.

For explicit detachment, the parameters of the detach method are each treated as roots for the purpose of determining the detached object graph. The fetch plan is applied to each of the roots as if no other roots were also being detached. The roots and their corresponding object graphs are combined and the resulting object graph is detached in its entirety.

12.7.1 Fetch Groups

Fetch groups are used to identify the list of fields and their associated field `recursion-depth` for each class for which the fetch plan is applied.

Fetch groups are identified by name and apply to one or more classes. Names have global scope so the same fetch group name can be used for any number of classes. This makes it possible to specify fetch groups per `PersistenceManager` instead of per extent. This greatly simplifies the use of fetch groups in an application.

The default fetch group (named "`default`") for each class is created by the JDO implementation according to the rules in the JDO 1.0.1 specification. That is, it includes all fields that by default belong to the default fetch group (i.e. single-valued fields), and causes the `jdoPostLoad` method to be called the first time fields are loaded. The default fetch group may also be defined by the user in the metadata like any other fetch group, in order to make use of JDO 2 features.

The implementation must also define another fetch group named "`all`" for each class. The "`all`" group contains all fields in the class, but can be redefined by the user, for example to add recursion-depth to certain fields, or to exclude some fields from being loaded.

If a fetch plan other than the default fetch plan is active for a `PersistenceManager`, the behavior of several APIs changes:

- For `detachCopy` the JDO implementation must ensure that the graph specified by the active fetch groups is copied, based on the `DETACH_LOAD_FIELDS` and `DETACH_UNLOAD_FIELDS` flags.
- For `refresh,` after clearing fields in the instances, the JDO implementation uses the fetch plan to determine which fields to load from the datastore.
- For `retrieve` with `FGonly` true, the implementation uses the fetch plan to determine which fields are loaded from the datastore. With `FGonly` false, the implementation reverts to JDO 1 behavior, which loads all fields from the datastore; in this case, no related instances are loaded.
- When executing a query the JDO implementation loads the fields as specified in the fetch plan associated with the `Query` instance.
- When the application dereferences an unloaded field, the JDO implementation uses the current fetch plan and the load-fetch-group of the field to create the fetch strategy for the field. The specific behavior depends on whether the unloaded field is a relation to another persistence-capable class.
  - for non-relation fields, the current fetch plan is applied to the field’s owning instance, and the fields in the field’s load-fetch-group, plus the field itself are added to the list of fields.
  - for relation fields, the fields in the owning instance are fetched as immediately above, and additionally the instances referred by the field are loaded using the current fetch plan plus the field’s load-fetch-group.

```
FetchPlan getFetchPlan();
```

This method retrieves the fetch plan associated with the `PersistenceManager`. It always returns the identical instance for the same `PersistenceManager`.

12.7.2 MaxFetchDepth

When relationship fields are included in the active fetch plan, it may be possible to retrieve the entire contents of the datastore, which might not be the desired effect. To avoid this behavior, and to allow the application to control the amount of data retrieved from the datastore, the `MaxFetchDepth` property of the fetch plan is used. The `MaxFetchDepth` is the depth of references (fields of relationship types) to instantiate, starting with the root instances.

Setting `MaxFetchDepth` to 1 limits the instances retrieved to the root instances and instances directly reachable from the root instances through a field in the fetch plan for the root class(es). Setting `MaxFetchDepth` to 0 has no meaning, and JDOUserException will be thrown. Setting `MaxFetchDepth` to -1 does not limit the instances retrieved via relationship fields in the fetch plan. Caution should be exercised to avoid retrieving more instances than desired.

For example, assume the class `Employee` defines field `dept` of type `Department`, and class `Department` defines field `comp` of type `Company`. When a query for `Employee` is executed, with a fetch plan that includes `Employee.dept` and `Department.comp` and with `MaxFetchDepth` set to 1, the `Departments` referenced by `Employees` returned from the query are instantiated, but the `Company` field is not instantiated. With the `MaxFetchDepth` set to 2, `Departments` and their corresponding `Companys` are instantiated for the `Employee` instances returned by the query.

12.7.3 Root instances

Root instances are parameter instances for `retrieve`, `detachCopy`, and `refresh`; result instances for queries. Root instances for `DetachAllOnCommit` are defined explicitly by the user via the `FetchPlan` property `DetachmentRoots` or `DetachmentRootClasses`. If not set explicitly, the detachment roots consist of the union of all root instances of methods executed since the last commit or rollback.

Once set explicitly, the detachment roots will not be changed until commit, at which time the detachment roots will be set to the empty collection.

Detachment roots and root classes are ignored for all `FetchPlans` except those associated directly with the `PersistenceManager`. Detachment root classes are never changed by the JDO implementation; they are completely controlled by the user. Detachment root classes is an empty `Class[]` when the `PersistenceManager` is first acquired from the `PersistenceManagerFactory`.

12.7.4 Recursion-depth

For object models with bidirectional relationships or self-referencing relationships, it is useful to limit the depth of the object graph retrieved through these relationships recursively. The recursion-depth attribute of the field element is used for this purpose. The recursion-depth for a relationship field specifies the number of times an instance of the same class, subclass, or superclass can be fetched via traversing this field.

A value of -1 means that the recursion-depth is not limited by traversing this field. If a field is defined in multiple fetch groups, the recursion-depth is the largest of the values specified, treating -1 as a very large positive number. If not specified in any fetch group or in the base field definition, the default is 1.

For example, assume a class `Directory` with a field `parent` of type `Directory` and a field `children` of type `Set<Directory>`, and assume the `recursion-depth` of the parent field is set to -1 and the `recursion-depth` of the `children` field is set to 2. When a query for a `Directory` is executed, all parents of the selected `Directory` instances will be retrieved, and all of the parents’ parents until a parent is found with a null parent. Additionally, all children of the selected `Directory` will be retrieved and all children of the children of the selected `Directory`.

12.7.5 The FetchPlan interface

Fetch groups are activated using methods on the interface `FetchPlan`. `PersistenceManager` and `Query` have `getFetchPlan()` methods. When a `Query` is retrieved from a `PersistenceManager`, its `FetchPlan` is initialized to the same settings as that of the `PersistenceManager`. Subsequent modifications of the `Query` `FetchPlan` are not reflected in the `FetchPlan` of the `PersistenceManager`. When an `Extent` is created, the `FetchPlan` of the `PersistenceManager` initializes the `FetchPlan` for the `Extent`.

Mutating `FetchPlan` methods return the `FetchPlan` instance to allow method chaining.

```
package javax.jdo;
```

```
public interface FetchPlan {
String DEFAULT = “default”;
String ALL = “all”;
int FETCH_SIZE_GREEDY = -1;
int FETCH_SIZE_OPTIMAL = 0;
int DETACH_LOAD_FIELDS = 1;
int DETACH_UNLOAD_FIELDS = 2;
/** Add the fetchgroup to the set of active fetch groups. Duplicate
names will be removed.*/
FetchPlan addGroup(String fetchGroupName);
/** Remove the fetch group from the set active fetch groups. */
FetchPlan removeGroup(String fetchGroupName);
/** Remove all active groups, including the default fetch group. */
FetchPlan clearGroups();
/** Return an immutable Set of the names of all active fetch groups.
*/
Set getGroups();
/** Set a Collection of group names to replace the current groups.
Duplicate names will be removed.*/
FetchPlan setGroups(Collection fetchGroupNames);
/** Set an array of group names to replace the current groups. Du-
plicate names will be removed.*/
FetchPlan setGroups(String[] fetchGroupNames);
/** Set a single group to replace the current groups. */
FetchPlan setGroup(String fetchGroupName);
/** Set the roots for DetachAllOnCommit */
FetchPlan setDetachmentRoots(Collection roots);
/** Get the roots for DetachAllOnCommit */
Collection getDetachmentRoots();
/** Set the roots for DetachAllOnCommit */
FetchPlan setDetachmentRootClasses(Class[] rootClasses);
/** Get the roots for DetachAllOnCommit */
Class[] getDetachmentRootClasses();
/** Set the maximum fetch depth. */
FetchPlan setMaxFetchDepth(int fetchDepth);
/** Get the maximum fetch depth. */
int setMaxFetchDepth();
/** Set the fetch size for large result set support. */
FetchPlan setFetchSize(int fetchSize);
/** Return the fetch size; 0 if not set; -1 for greedy fetching. */
```

```
int getFetchSize();
/** Set detachment options */
FetchPlan setDetachmentOptions(int options);
/** Return the detachment options */
int getDetachmentOptions();
```

The `getGroups` method returns a collection of names. After a call to `clearGroups()` this method returns an empty `Set`. It is legal to remove the default fetch group explicitly via `pm.getFetchPlan().removeGroup("default")`, or to use `setGroups()` with a collection that does not contain "`default`". This makes it possible to have only a given fetch group active without the default fetch group. If no fetch groups are active then a `Set` with no elements is returned. In this case, loading an instance might not result in loading the default fetch group fields and the `jdoPostLoad` method will only be called if there is an active fetch group that declares `post-load=”true”`.

The fetch size allows users to explicitly control the number of instances retrieved from queries. A positive value is the number of result instances to be fetched. A value of `FETCH_SIZE_GREEDY` indicates that all results should be obtained immediately. A value of `FETCH_SIZE_OPTIMAL` indicates that the JDO implementation should try to optimize the fetching of results.

Note that the graph and fields specified by a `FetchPlan` is strictly the union of all the active fetch groups not based on any complicated set mathematics. So, if a field f1 is in fetch groups A and B, and both A and B are added to the `FetchPlan,`and subsequently B is removed from the active fetch groups and the instance is loaded, then the field f1 will be loaded, because it is in fetch group A.

Examples:

```
pm = pmf.getPersistenceManager();
FetchPlan fp = pm.getFetchPlan();

fp.addGroup("detail").addGroup("list");
// prints [default, detail, list]
System.out.println(fp.getGroups());
// refreshes fields in any of default+detail+list
pm.refresh(anInstance);

fp.clearGroups();
// prints []
System.out.println(fp.getGroups());
pm.refresh(anInstance); // doesn’t do anything

fp.addGroup("list");
// prints [list]
System.out.println(fp.getGroups());
// refreshes fields in list only
pm.refresh(anInstance);
```

When an instance is loaded using `getObjectById`, a `Query` is executed, or an `Extent` is iterated, the implementation may choose to use the active fetch groups to prefetch data. If an instance being loaded does not have a fetch group with the same name as any of the active groups, and the semantics of the method allow returning a hollow instance, then it may be loaded as hollow. If it has more than one of the active groups then the union of fields in all active groups is used.

Instances loaded through field navigation behave in the same way as for `getObjectById` except that an additional fetch group may be specified for the field in the metadata using the new "`load-fetch-group`" attribute. If present the load-fetch-group is considered active just for the loading of the field. This can be used to load several fields together when one of them is touched. The field touched is loaded even if it is not in the load-fetch-group.

For the `refresh` and `retrieve` methods, the implementation must ensure that only the graph specified by the active fetch groups is refreshed or retrieved; i.e. these operations will recursively refresh or retrieve the instances and fields in the graph covered by the active fetch groups. The refreshed or retrieved graph must not contain extra instances but extra fields may be refreshed for an instance in the graph.

12.7.6 Defining fetch groups

Fetch groups are only defined in the metadata for a class or interface.

```
<!ELEMENT fetch-group (extension*,(fetch-group|field|property)*,
extension*)>
<!ATTLIST fetch-group name CDATA #REQUIRED>
<!ATTLIST fetch-group post-load (true|false) #IMPLIED>
<!ATTLIST field recursion-depth CDATA #IMPLIED>
<!ATTLIST property recursion-depth CDATA #IMPLIED>
```

The `post-load` attribute on the `fetch-group` element indicates whether the `jdoPostLoad` callback will be made when the fetch group is loaded. It defaults to `false`, for all fetch groups except the default fetch group, on which it defaults to `true`. The callback will be called if any field of an instance is loaded when any fetch group is active that contains the `post-load` attribute set to `true`.

The `name` attribute on a `field` element contained within a `fetch-group` element is the name of field in the enclosing class, or a dot-separated expression identifying a field reachable from the class by navigating a reference, a collection, or a map. For maps of persistence-capable classes "#key" or "#value" may be appended to the name of the map field to navigate the key or value respectively (e.g. to include a field of the key class or value class in the fetch group).

For collection and arrays of persistence-capable classes, "`#element`" may be appended to the name of the field to navigate the element. This is optional; if omitted for collections and arrays, `#element` is assumed.

Recursive fetch group references are controlled by the `recursion-depth` attribute on a contained field or property element of a fetch-group. A `recursion-depth` of 0 will fetch the whole graph of instances reachable from this field. The default is 1, meaning that only the instance directly reachable from this field is fetched.

A contained `fetch-group` element indicates that the named group is to be included in the group being defined. Nested fetch group elements are limited to only the name attribute and no contained elements. That is, it is not permitted to nest entire fetch group definitions. If there are two definitions for a reference, collection or map field (due to fetch groups including other fetch groups) then the union of the fetch groups involved is used. If one or more depths have been specified then the largest depth is used unless one of the depths has not been specified (unlimited overrides other depth specifications).

```
public class Person {
  private String name;
  private Address address;
  private Set children;
}

public class Address {
  private String street;
  private String city;
  private Country country;
}

public class Country {
  private String code;
  private String name;
}

<class name="Person" ...>
...
  <!-- name + address + country code -->
  <fetch-group name="detail">
    <fetch-group name="default"/>
    <field name="address"/>
    <field name="address.country.code"/>
  </fetch-group>

  <!-- name + address + country code + same for children -->
  <fetch-group name="detail+children">
    <fetch-group name="detail"/>
    <field name="children"/>
  </fetch-group>

  <!-- name + address + country code + names of children -->
  <fetch-group name="detail+children-names">
    <fetch-group name="detail"/>
    <field name="children#element.name"/>
  </fetch-group>

  <!-- name + address + country code + list fg of children -->
  <fetch-group name="detail+children-list">
    <fetch-group name="detail"/>
    <field name="children" fetch-group="list"/>
  </fetch-group>

</class>
```

Here is a map example:

```
public class Node {
  private String name;
  private Map edges; // Node -> EdgeWeight
}

public class EdgeWeight {
  private int weight;
}

<class name="Node" ...>
  ...
  <fetch-group name="neighbour-weights">
    <field name="edges#key.name"/>
    <field name="edges#value"/>
  </fetch-group>
  <fetch-group name="neighbours">
    <field name="edges"/>
  </fetch-group>
  <fetch-group name="whole-graph">
    <field name="edges" fetch-depth=”0”/>
  </fetch-group>
</class>
```

12.8 Flushing instances

```
void flush();
```

This method flushes all dirty, new, and deleted instances to the datastore. It has no effect if a transaction is not active.

If a datastore transaction is active, this method synchronizes the cache with the datastore and reports any exceptions.

If an optimistic transaction is active, this method obtains a datastore connection and synchronizes the cache with the datastore using this connection. The connection obtained by this method is held until the end of the transaction.

```
void checkConsistency();
```

This method validates the cache with the datastore. It has no effect if a transaction is not active.

If a datastore transaction is active, this method verifies the consistency of instances in the cache against the datastore. An implementation might flush instances as if) were called, but it is not required to do so.

If an optimistic transaction is active, this method obtains a datastore connection and verifies the consistency of the instances in the cache against the datastore. If any inconsistencies are detected, a `JDOOptimisticVerificationException` is thrown. This exception contains a nested `JDOOptimisticVerificationException` for each object that failed the consistency check. No datastore resources acquired during the execution of this method are held beyond the scope of this method.

12.9 Transaction completion

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

12.10 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.

12.11 User associated objects

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.For applications where multiple users need to access their own user objects, the following methods allow user objects to be stored and retrieved by key. The values are not examined by the `PersistenceManager`.

There are no restrictions on values. Keys must not be null. For proper behavior, the keys must be immutable (e.g. `java.lang.String`, `java.lang.Integer`, etc.) or the keys' identity (to the extent that it modifies the behavior of equals and hashCode methods) must not change while a user object is associated with the key. This behavior is not enforced by the `PersistenceManager`.

`Object putUserObject(Object key, Object value);`

This method models the `put` method of `Map`. The current value associated with the key is returned and replaced by the parameter value. If the parameter `value` is `null`, the implementation may remove the entry from the table of managed key/value pairs.

`Object removeUserObject(Object key);`

This method models the `remove` method of `Map`. The current value associated with the key is returned and removed.

`Object getUserObject(Object key);`

This method models the `get` method of `Map`. The current value associated with the key is returned. If the `key` is not found in the table, `null` is returned.

12.12 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();`This methos returns the `PersistenceManagerFactory` that created this `PersistenceManager`.

12.13 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, Object key);
```

This method returns an object id instance corresponding to the `pcClass` and `key` arguments. A `String` argument might have been the result of executing `toString` on an object id instance. The `key` argument is the value of the key field for single field identity.

This method is portable for datastore identity and application identity.

12.14 Sequence

The JDO metadata defines named sequence value object generators, or simply, sequences. A sequence implements the `javax.jdo.datastore.Sequence` interface.

The behavior of the sequence with regard to transactions and rolling over maximum values is specified in the metadata.

Note that there is no portable way for a user-defined sequence to implement the `Sequence` interface. In particular, the `getName` method might not return the name of the sequence, and the transactional behavior of the sequence as specified by the user in metadata might not be implemented. A future version of the specification might add a sequence factory spi to enable portable user-defined sequences.

The `PersistenceManager` provides a method to retrieve a `Sequence` by name.

```
Sequence getSequence(String name);
```

If the named sequence does not exist, `JDOUserException` is thrown.

The name is the scoped name of the sequence , which uses the standard Java package naming. For example, a sequence might be named “`com.acme.hr.EmployeeSequence`”.

```
package javax.jdo.datastore;
public interface Sequence {
   String getName();
```

This method returns the fully qualified name of the `Sequence`.

```
   Object next();
```

This method returns the next sequence value object. The sequence might be protected by transactional semantics, in which case the sequence value object will be reused if the transaction in which the sequence value object was obtained rolls back.

```
   void allocate(int additional);
```

This method is a hint to the implementation that the application needs the additional number of sequence value objects in short order. There is no externally visible behavior of this method. It is used to potentially improve the efficiency of the algorithm of obtaining additional sequence value objects.

```
   Object current();
```

This method returns the current sequence value object if it is available. It is intended to return a sequence value object previously used The implementation might choose to return `null` for all cases or for any cases where a current sequence value object is not available.

```
   long nextValue();
```

This method returns the next sequence value as a `long` if it is available and able to be converted to a number. It is equivalent to `((Long)next()).longValue().`

```
   long currentValue();
```

This method returns the current sequence value as a `long` if it is available and able to be converted to a number. It is equivalent to `((Long)current()).longValue().`

}

12.15 Life-cycle callbacks

In order to minimize the impact on domain classes, the instance callbacks can be defined to use a life-cycle listener pattern instead of having the domain class implement the callback interface(s).

```
package javax.jdo.listener;
public interface InstanceLifecycleListener {
}
public interface CreateLifecycleListener
   extends InstanceLifecycleListener {
void postCreate(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is created, during `makePersistent`. It is called after the instance transitions to persistent-new.

```
package javax.jdo.listener;
public interface LoadLifecycleListener
   extends InstanceLifecycleListener {
void postLoad(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is loaded. It is called after the `jdoPostLoad` method is invoked on the instance.

```
package javax.jdo.listener;
public interface StoreLifecycleListener
   extends InstanceLifecycleListener {
void preStore(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is stored, for example during flush or commit. It is called before the `jdoPreStore` method is invoked on the instance. An object identity for a persistent-new instance might not have been assigned to the instance when this callback is invoked.

```
void postStore(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is stored, for example during flush or commit. It is called after the `jdoPreStore` method is invoked on the instance. An object identity for a persistent-new instance must have been assigned to the instance when this callback is invoked.

```
package javax.jdo.listener;
public interface ClearLifecycleListener
   extends InstanceLifecycleListener {
void preClear(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is cleared, for example during `afterCompletion`. It is called before the `jdoPreClear` method is invoked on the instance.

```
void postClear(InstanceLifecycleEvent event);
```

This method is called whenever a persistent instance is cleared, for example during `afterCompletion`. It is called after the `jdoPreClear` method is invoked on the instance and the fields have been cleared by the JDO implementation.

```
package javax.jdo.listener;
public interface DeleteLifecycleListener
   extends InstanceLifecycleListener {
void preDelete(InstanceLifecycleEvent event);
```

This method is called whenever a persistent instance is deleted, during `deletePersistent`. It is called before the state transition and before the `jdoPreDelete` method is invoked on the instance.

```
void postDelete(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is deleted, during `deletePersistent`. It is called after the `jdoPreDelete` method is invoked on the instance and after the state transition.

```
package javax.jdo.listener;
public interface DirtyLifecycleListener
   extends InstanceLifecycleListener {
void preDirty(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent clean instance is first made dirty, during an operation that modifies the value of a persistent or transactional field. It is called before the field value is changed. During this method, the instance responds `false` to `isDirty`. During this method, fields in the source instance and others might be changed, but this method will only be invoked once until the instance is no longer dirty.

```
void postDirty(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent clean instance is first made dirty, during an operation that modifies the value of a persistent or transactional field. It is called after the field value was changed. During this method, the instance responds `true` to `isDirty`. During this method, fields in the source instance and others might be changed, but this method will only be invoked once until the instance is no longer dirty.

```
package javax.jdo.listener;
public interface DetachLifecycleListener
   extends InstanceLifecycleListener {
```

```
void preDetach(InstanceLifecycleEvent event);
}
```

This method is called before a persistent instance is copied for detachment. It is called before the `jdoPreDetach` callback.

```
void postDetach(InstanceLifecycleEvent event);
}
```

This method is called whenever a persistent instance is copied for detachment. The source instance is the detached copy; the target instance is the persistent instance. It is called after the `jdoPostDetach` callback on the detached copy.

```
package javax.jdo.listener;
public interface AttachLifecycleListener
   extends InstanceLifecycleListener {
void preAttach(InstanceLifecycleEvent event);
}
```

This method is called before a detached instance is attached, via the `makePersistent` method. The source instance is the detached instance. This method is called before the corresponding `jdoPreAttach` on the detached instance.

```
void postAttach(InstanceLifecycleEvent event);
}
```

This method is called after a detached instance is attached. The source instance is the corresponding persistent instance in the cache; the target instance is the detached instance. This method is called after the corresponding `jdoPostAttach` on the persistent instance.

InstanceLifecycleEvent

This class is provided as part of the javax.jdo.listener package.

Note that although `InstanceLifecycleEvent` inherits `Serializable` interface from `EventObject`, it is not intended to be `Serializable`. Appropriate serialization methods are implemented to throw `NotSerializableException`.

```
package javax.jdo.listener;
public class InstanceLifecycleEvent
   extends java.util.EventObject {
static final int CREATE = 0;
static final int LOAD = 1;
static final int STORE = 2;
static final int CLEAR = 3;
static final int DELETE = 4;
static final int DIRTY = 5;
static final int DETACH = 6;
static final int ATTACH = 7;
int getEventType();
```

This method returns the event type that triggered the event.

```
InstanceLifecycleEvent(int type, Object source);
```

This constructor creates an instance with the type, and source object.

```
InstanceLifecycleEvent(int type, Object source, Object target);
```

This constructor creates an instance with the type, source, and target objects.

```
Object getSource();
```

This method returns the object for which the event was triggered. This method is inherited from the `EventObject` class.

```
Object getTarget();
```

This method returns the “other” object associated with the event. Specifically, the target object is the detached instance in the case of `postAttach`, and the persistent instance in the case of `postDetach`. The target must be null for all other cases.

```
Object getPersistentInstance();
```

This method returns the persistent instance for which the event was triggered. This method is a convenience method that returns the source or target depending on the event.

```
Object getDetachedInstance();
```

This method returns the detached instance for which the event was triggered. This method is a convenience method that returns the source or target depending on the event.

```
}
```

```
void addInstanceLifecycleListener (InstanceLifecycleListener listener, Class[] classes);
```

This `PersistenceManager` method adds the listener to the list of lifecycle event listeners. The classes parameter identifies all of the classes of interest. If the classes parameter is specified as `null`, events for all persistent classes and interfaces are generated. If the classes specified have persistence-capable subclasses, all such subclasses are registered implicitly.

The listener will be called for each event for which it implements the corresponding listener interface.

```
void removeInstanceLifecycleListener (InstanceLifecycleListener listener);
```

This `PersistenceManager` method removes the listener from the list of event listeners.

12.16 Access to internal datastore connection

In order for the application to perform some datastore-specific functions, such as to execute a query that is not directly supported by JDO, applications might need access to the datastore connection used by the JDO implementation. This method returns a wrapped connection that can be cast to the appropriate datastore connection and used by the application.

The capability to get the datastore connection is indicated by the optional feature string `javax.jdo.option.GetDataStoreConnection`.

```
package javax.jdo.datastore;
public interface JDOConnection {
   Object getNativeConnection();
   void close();
}
```

`JDOConnection getDataStoreConnection();`

If this method is called while a datastore transaction is active, the object returned will be enlisted in the current transaction. If called in an optimistic transaction before flush has been called, or outside an active transaction, the object returned will not be enlisted in any transaction.

The object must be returned to the JDO implementation prior to calling any JDO method or performing any action on any persistent instance that might require the JDO implementation to use a connection. If the object has not been returned and the JDO implementation needs a connection, a `JDOUserException` is thrown. The object is returned to the JDO implementation by calling the standard method on the object.

For JDOR implementations

- the `JDOConnection` obtained by `getDataStoreConnection` implements `java.sql.Connection`.
- The application returns a JDBC Connection to the JDO implementation by calling its `close()` method.

SQL Portability

For portability, a JDBC-based JDO implementation will return an instance that implements `java.sql.Connection`. The instance will throw an exception for any of the following method calls: commit, getMetaData, releaseSavepoint, rollback, setAutoCommit, setCatalog, setHoldability, setReadOnly, setSavepoint, setTransactionIsolation, and setTypeMap.

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 `PersistenceManager`s 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

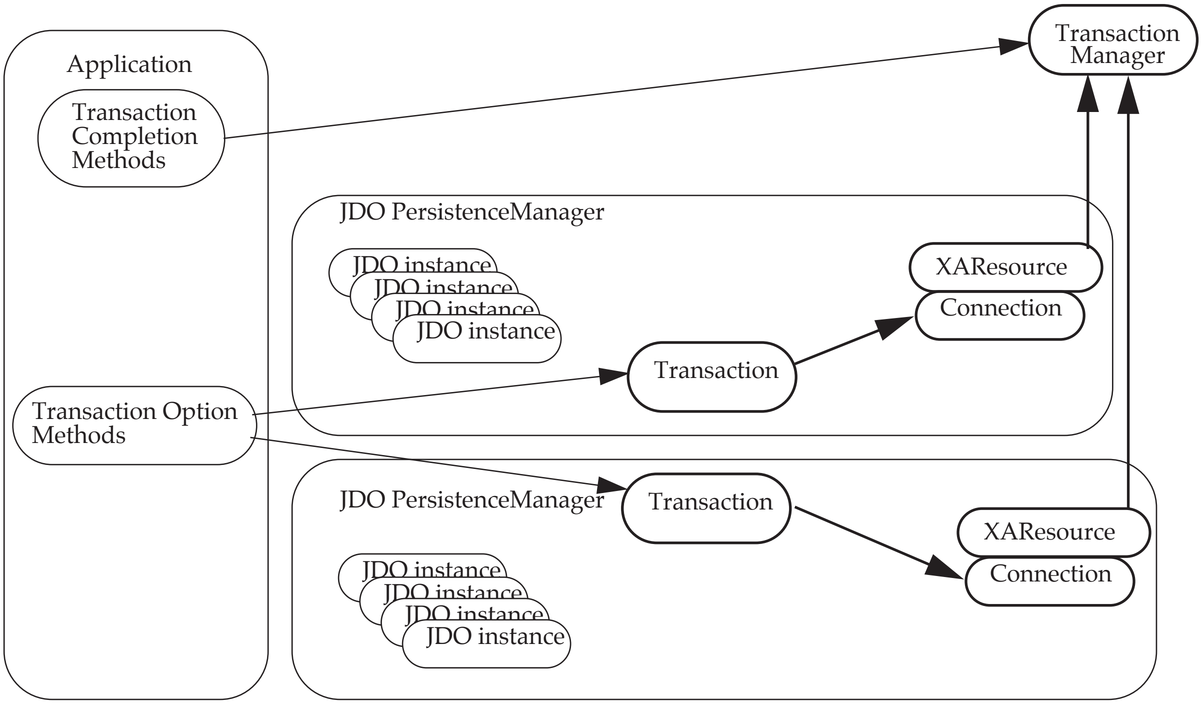
There are two types of transaction management strategies supported by JDO: "datastore transaction management"; and "optimistic transaction management".

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 16.0 Transactions and Connections



13.4 Interface Transaction

```
package javax.jdo;
public 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 a return value of `true` means that 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 before the `afterCompletion` synchronization method is called. The method returns `false` during `afterCompletion`.

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` synchronization method), a `JDOUserException` is thrown. These methods can be called during `afterCompletion` processing.

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 read access (including navigation) are allowed without an active transaction. If this flag is set to `false`, then queries and non-primary key 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.

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.

This flag is only used if the `PersistenceManager` `DetachAllOnCommit` flag is `false`.

`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.

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 replaced. 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 two `Synchronization` 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).

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`.

During transaction completion, the environment calls the jdo implementation's `beforeCompletion` method, which in turn calls the user's `beforeCompletion` method registered by the `setSynchronization` method.

During the user's `beforeCompletion` method, fields in persistent and transactional instances might be changed, persistent instances might be deleted, and instances might be made persistent. These changes will be reflected in the current transaction.

After the user's `beforeCompletion` method completes, the jdo implementation flushes the cache to the datastore. During flush, life cycle methods declared in the persistence-capable classes are called back, as well as methods on instances registered with the `PersistenceManager` via `addInstanceLifecycleListener`.

After transaction completion, the environment calls the jdo implementation's `afterCompletion` method, which performs state transitions of the instances in the cache. During these state transitions, life cycle methods declared in the persistence-capable classes are called back, as well as methods on instances registered with the `PersistenceManager` via `addInstanceLifecycleListener`. Subsequently, the jdo implementation calls the user's `afterCompletion` method registered by the `setSynchronization` method. The parameter for the `afterCompletion(int status)` method will be either `javax.transaction.Status.STATUS_COMMITTED` or `javax.transaction.Status.STATUS_ROLLEDBACK`.

```
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 `PersistenceManager`s.

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 `begin`, `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.

If `commit` or `rollback` is called when a transaction is not active, `JDOUserException` is thrown. If `begin` is called when a transction is active, `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:

- rolls back changes made in this transaction from the datastore;
- transitions persistent instances according to the life cycle specification;
- 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.

13.4.5 RollbackOnly

At times, a component needs to mark a transaction as failed even though that component is not authorized to complete the transaction. In order to mark the transaction as unsuccessful, and to determine if a transaction has been so marked, two methods are used:

```
void setRollbackOnly();
boolean getRollbackOnly();
```

Either the user application or the JDO implementation may call `setRollbackOnly`. There is no way for the application to determine explicitly which component called the method.

Once a transaction has been marked for rollback via `setRollbackOnly`, the commit method will always fail with `JDOFatalDataStoreException`. The JDO implementation must not try to make any changes to the database during commit when the transaction has been marked for rollback.

When a transaction is not active, and after a transaction is begun, `getRollbackOnly` will return `false`. Once `setRollbackOnly` has been called, it will return `true` until `commit` or `rollback` is called.

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. This chapter specifies the query language “JDOQL”, and includes conventions for the use of “SQL” as the language for JDO implementations using a relational store.

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.
- Deletion by query. Deleting multiple instances in the datastore can be done efficiently if specified as a query method instead of instantiating all persistent instances and calling the `deletePersistent` method on them.

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. If the class is not explicitly passed to the query, it is obtained from the Extent.
- 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. If the collection is not explicitly passed to the query, then it is obtained from the class.
- 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 `Object`s, 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. To be portable, the expression's type must be one of:
  - primitive types except `boolean`;
  - wrapper types except `Boolean`;
  - `BigDecimal`;
  - `BigInteger`;
  - `String`;
  - `Date`.
- result specification. The application might want to get results from a query that are not instances of the candidate class. The results might be fields of persistent instances, instances of classes other than the candidate class, or aggregates of fields.
- grouping specification. Aggregates are most useful when the application can specify the result field by which to group the results.
- uniqueness. The application can specify that the result of a query is unique, and therefore a single value instead of a `Collection` should be returned from the query.
- result class. The application may have a user-defined class that best represents the results of a query. In this case, the application can specify that instances of this class should be returned.
- limiting the size of the results. The application might want to limit the number of instances returned by the query, and might want to skip over some number of instances that might have been returned previously.

The class implementing the `Query` interface must be serializable. The serialized fields include the candidate class, the filter, parameter declarations, variable declarations, imports, ordering specification, uniqueness, result specification, grouping specification, and result class. The candidate collection, limits on size, and number of skipped instances are not serialized. 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`, `setGrouping`, `setOrdering`, and `setResult` correspond to the method body and do not introduce names to the namespace

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

Keywords

Keywords must not be used as package names, class names, parameter names, or variable names in queries. Keywords are permitted as field names only if they are on the right side of the "." in field access expressions as defined in the Java Language Specification second edition, section 15.11. Keywords include the Java language keywords and the JDOQL keywords. Java keywords are as defined in the Java language specification section 3.9, plus the boolean literals true and false, and the null literal. JDOQL keywords are the following:

select, SELECT, unique, UNIQUE, distinct, DISTINCT, avg, AVG, min, MIN, max, MAX, count, COUNT, sum, SUM, as, AS, into, INTO, from, FROM, exclude, EXCLUDE, subclasses, SUBCLASSES, where, WHERE, order, ORDER, by, BY, ascending, ASCENDING, asc, ASC, descending, DESCENDING, desc, DESC, group, GROUP, having, HAVING, parameters, PARAMETERS, variables, VARIABLES, range, RANGE.

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 packages `java.lang` and `javax.jdo` are automatically imported as if the declaration "`import java.lang.*; import javax.jdo.*;`" 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 in the filter preceded by ":" has the same effect. A parameter name hides the name of a candidate class field if equal. Parameter names must be unique.

The method `declareVariables` introduces the names of 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 name in the filter that is not a parameter name or a field name is implicitly a variable name.

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 a new empty query instance.

`Query newQuery (Object query);`

Construct a new query instance from another query instance. JDO implementations must support a serialized/restored `Query` instance from the same JDO vendor but a different execution environment, a query instance currently bound to the same `PersistenceManager`, and a query instance currently bound to a `PersistenceManager` from the same JDO vendor. Any of the elements Class, QueryString, IgnoreCache flag, Result, ResultClass, Import declarations, Variable declarations, Parameter declarations, Grouping, 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 queryString);`

Construct a new query instance using the specified `String` as the single-string representation of the query [see section 14.6.13].

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

Construct a new 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`”. In this case, the parameter is a `String` representing the single-string version of the query [see section 14.6.13].

For use with SQL, the language parameter is “`javax.jdo.query.SQL`” and the query parameter is a `String` containing the SQL query [see section 14.7]. Other languages’ parameter is not specified.

`Query newQuery (Class cls);`

Construct a new query instance with the candidate class specified.

`Query newQuery (Extent cln);`

Construct a new query instance with the candidate `Extent` specified; the candidate class is taken from the `Extent`.

`Query newQuery (Class cls, Collection cln);`

Construct a new query instance with the candidate class and candidate `Collection` specified.

`Query newQuery (Class cls, String queryString);`

Construct a new query instance with the candidate class and query string specified.The query string parameter might be the filter or the single string representing the query [see section 14.6.13].

`Query newQuery (Class cls, Collection cln, String queryString);`

Construct a query instance with the candidate class, the candidate `Collection`, and query string specified.The query string parameter might be the filter or the single string representing the query [see section 14.6.13].

`Query newQuery (Extent cln, String queryString);`

Construct a new query instance with the candidate `Extent` and query string specified; the candidate class is taken from the `Extent`.The query string parameter might be the filter or the single string representing the query [see section 14.6.13].

`Query newNamedQuery (Class cls, String queryName);`

Construct a new query instance with the given candidate class from a named query. The query name given must be the name of a query defined in metadata. The metadata is searched for the specified name. The extent, including subclasses, is the default for the candidate collection.

If the named query is not found in already-loaded metadata, the query is searched for using an algorithm. Files containing metadata are examined in turn until the query is found. The order is based on the metadata search order for class metadata, but includes files named based on the class name.

The file search order for a query scoped to class com.sun.nb.Bar is: META-INF/package.jdo, WEB-INF/package.jdo, package.jdo, com/package.jdo, com/sun/package.jdo, com/sun/nb/package.jdo, com/sun/nb/Bar.jdo. Once metadata for the class is found, no more .jdo files will be examined for the class.

If the metadata is not found in the above, and there is a property in the PersistenceManagerFactory javax.jdo.option.Mapping=mySQL, then the folowing files are searched: META-INF/package-mySQL.orm, WEB-INF/package-mySQL.orm, package-mySQL.orm, com/package-mySQL.orm, com/sun/package-mySQL.orm, com/sun/nb/package-mySQL.orm, com/sun/nb/Bar-mySQL.orm. Once mapping metadata for the class is found, no more .orm files will be examined for the class.

If metadata is not found in the above, then the following files are searched: META-INF/package.jdoquery, WEB-INF/package.jdoquery, package.jdoquery, com/package.jdoquery, com/sun/package.jdoquery, com/sun/nb/package.jdoquery, com/sun/nb/Bar.jdoquery. Once the query metadata is found, no more .jdoquery files will be examined for the query.

If the metadata for the named query is not found in the above, a `JDOUserException` is thrown.

NOTE: If no class is provided as a parameter, the metadata must be in one of the top level locations or must have already been processed during loading of metadata for a class or interface whose metadata has been loaded.

This resource name is loaded by one of the three class loaders used to resolve resource names (see Section 12.5). The loaded resource must contain the metadata definition of the query name. The schema for the loaded resource is the same as for the .jdo file.

If the `unmodifiable` attribute is specified as or defaults to `“false”`, then the `Query` instance returned from this method can be modified by the application, just like any other `Query` instance.

Named queries must be compilable. Attempts to get a named query that cannot be compiled result in `JDOUserException`.

14.6 Query Interface

```
package javax.jdo;
public interface Query extends Serializable {
String JDOQL = “javax.jdo.query.JDOQL”;
String SQL = “javax.jdo.query.SQL”;
```

Persistence Manager

```
PersistenceManager getPersistenceManager();
```

Return the associated `PersistenceManager` instance. If this `Query` instance was restored from a serialized form, then `null` is returned.

Fetch Plan

```
FetchPlan getFetchPlan();
```

This method retrieves the fetch plan associated with the `Query`. It always returns the identical instance for the same `Query` instance. Any change made to the fetch plan affects subsequent query execution. Fetch plan is described in Section 12.7.

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 types and names 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.

`void setResult (String result);`

Specify the results of the query if not instances of the candidate class.

`void setGrouping (String grouping);`

Specify the grouping of results for aggregates.

`void setUnique (boolean unique);`

Specify that there is a single result of the query.

`void setResultClass (Class resultClass);`

Specify the class to be used to return result instances.

`setRange (long fromIncl, long toExcl);`

`setRange (String fromIncltoExcl);`

Specify the number of instances to skip over and the maximum number of result instances to return.

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 modification

`void setUnmodifiable();`

`boolean isUnmodifiable();`

The `Unmodifiable` option, when set, disallows further modification of the query, except for specifying the range, result class, and `ignoreCache` option.

Query evaluation

This discussion covers queries constructed by one of these methods: `newQuery(String singleStringQuery); newNamedQuery(Class candidateClass, String namedQueryName); or newQuery(“javax.jdo.query.JDOQL”, Object singleStringQuery)`.

- the candidate class cannot be overridden via `setClass` except where there is either an exact match of the name in the JDOQL query and the `setClass` parameter, or where the FROM clause is missing from the query string in the `newQuery` method.
- the single string query is first parsed to yield the result, result class, filter, variable list, parameter list, import list, grouping, ordering, and range.
- then, the values specified in APIs `setResult`, `setResultClass`, `setFilter`, `declareVariables`, `declareParamters`, `declareImports`, `setGrouping`, `setOrdering`, and `setRange` override the corresponding settings from the single string query.

Evaluation of implicit parameters and variable declarations is done after applying overrides from APIs.

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. By default, they return an unmodifiable `List` which the user can iterate to get results. The user can specify the class of the result of executing a query. Executing any operation on the `List` that might change it throws `UnsupportedOperationException`. The signature of the `execute` methods specifies that they return an `Object` that must be cast to the proper type 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, but this behavior is not portable. 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 `JDOFatalUserException`. If the `NontransactionalRead` 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 by default is an unmodifiable `List` of instances that satisfy the boolean filter. The result may be a large `List`, which should be iterated or possibly passed to another `Query`. The `size()` method returns `Integer.MAX_VALUE` if the actual size of the result is not known (for example, the `List` represents a cursored result); if the size of the result equals or exceeds `Integer.MAX_VALUE`; or if the range equals or exceeds `Integer.MAX_VALUE`.

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.
- modified instances will be evaluated using their current transactional values.

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 might or might not be considered part of the candidate instances.
- modified instances might be evaluated using their current transactional values or the values as they exist in the datastore, which might not reflect the current transactional values.

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 parameters);`

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.If implicit parameters are used, the keys in the map do not include the leading ":".

`Object executeWithArray (Object[] parameters);`

The `executeWithArray` method is similar to the `execute` method, but takes its parameters from an array instance. The array contains `Object`s, 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 `Employee`s 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`, `String`, and `Map` methods documented below. 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`.
- There is no distinction made between `char` literals and `String` literals. Single-character `String` literals can be used wherever char literals are permitted. `Char` literals will be widened if used in numerical expressions; or treated as single-character `String` literals if used in `String` expressions.
- String literals are allowed to be delimited by single quote marks or double quote marks. This allows String literal filters to use single quote marks instead of escaped double quote marks.

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 or public final static field names are required to be supported by JDO implementations. Other identifiers might be supported but are not required. Thus, portable queries must not use fields other than persistent or public final 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
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
&	boolean logical AND (not bitwise)
&&	conditional AND
|	boolean logical OR (not bitwise)
||	conditional OR
~	integral unary bitwise complement
+	binary addition, unary plus, or String concatenation
-	binary subtraction or unary numeric sign inversion
*	times
/	divide by
!	logical complement
%	modulo operator
instanceof	instanceof operator

- 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 persistence-capable types use the JDO Identity comparison of the references; this includes containment methods applied to `Collection` and `Map` types. 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-persistence-capable reference types uses the `equals` method of the type; this includes containment methods applied to `Collection` and `Map` types.
- `String` methods `startsWith` and `endsWith` support wild card queries but not in a portable way. JDO does not define any special semantic to the argument passed to the method; in particular, it does not define any wild card characters. To achieve portable behavior, applications should use `matches(String)`.

`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()`).Methods

The following methods are supported for their specific types, with semantics as defined by the Java language:

Table 5: Query Methods

Method	Description
contains(Object)	applies to Collection types
get(Object)	applies to Map types
containsKey(Object)	applies to Map types
containsValue(Object)	applies to Map types
isEmpty()	applies to Map and Collection types
size()	applies to Map and Collection types

Table 5: Query Methods

Method	Description
toLowerCase()	applies to String type
toUpperCase()	applies to String type
indexOf(String)	applies to String type; 0-indexing is used
indexOf(String, int)	applies to String type; 0-indexing is used
matches(String)	applies to String type; only the following regular expression patterns are required to be supported and are portable: global “(?i)” for case-insensitive matches; and “.” and “.*” for wild card matches. The pattern passed to matches must be a literal or parameter.
substring(int)	applies to String type
substring(int, int)	applies to String type
startsWith(String)	applies to String type
endsWith(String)	applies to String type
Math.abs(numeric)	static method in java.lang.Math, applies to types of float, double, int, and long
Math.sqrt(numeric)	static method in java.lang.Math, applies to double type
JDOHelper.getObjectId(Object)	static method in JDOHelper, allows using the object identity of an instance directly in a query.

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` or a `JDOUserException` is thrown.

Parameters must all be declared explicitly via `declareParameters` or all be declared implicitly in the filter. Parameters implicitly declared (in the result, filter, ordering, grouping, or range) are identified by prepending a ":" to the parameter everywhere it appears. All parameter types can be determined by one of the following techniques:

- the parameter is used as the right hand side or left hand side of a boolean operator (<, <=, ==, >=, or >) and the other side is strongly typed, or
- the parameter is used in a method from Table 5 on page 157 directly as either a parameter or the object on which the method is called, and the type can be determined from the context of the method, or
- the parameter is explicitly cast using the cast operator and the cast is identical everywhere the parameter appears.

Implicit parameter declaration

When parameters are declared implicitly, if the query is string-based, parameters are recognized in the order that they appear in the query string. If the query is API-based, parameters are recognized as if declared explicitly, with the order of their first appearance in the result, filter, grouping, ordering, and range. This is significant if a positional form of execute is used.

14.6.4 Import statements

The import statements follow the Java syntax for import statements. Import statements are separated by semicolons. Import on demand is supported. Classes in `java.lang` and `javax.jdo` are automatically imported.

14.6.5 Variable declaration

The type declarations follow the Java syntax for local variable declarations. Variable declarations are separated by semicolons.

A variable that is not constrained with an explicit contains clause is constrained by the extent of the persistence capable class (including subclasses). If the class does not manage an Extent, then no results will satisfy the query.

If the query result uses a variable, the variable must not be constrained by an extent. Further, each side of an "OR" expression must constrain the variable using a contains clause.

A portable query will constrain all variables with a `contains` clause in each side of an “OR” expression of the filter where the variable is used. Further, each variable must either be used in the query result or its `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.

The semantics of contains is “exists”, where the contains clause is used to filter instances. 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 variable is used in the result, then it need not be constrained.

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”. If a variable is used in the result, then it must not be used in a negated contains clause.

Implicit variable declaration

The variable declaration is unnecessary if all variables are implicitly declared. All variables must be explicitly declared, or all variables must be implicitly declared.

Names in the filter are treated as parameters if they are explicitly declared via `declareParameters` or if they begin with “:”.

Names are treated as variable names if they are explicitly declared via `declareVariables`.

Names are treated as field or property names if they are fields or properties of the candidate class.

Names are treated as class names if they exist in the package of the candidate class, have been imported, or if they are in the `java.lang` package. e.g. `Integer`.

Otherwise, names are treated as implicitly defined variable names.

Variables must be typed. Implicitly defined variables are typed according to the following:

- if the variable is the parameter of a `contains` method, the type is the element type of the collection; or
- if the variable is the parameter of a `containsKey` method, the type is the key type of the map; or
- if the variable is the parameter of a `containsValue` method, the type is the value type of the map; or
- if the variable is not constrained by a `contains`, `containsKey`, or `containsValue` method, the variable must be typed by an explicit cast the first time the variable appears in the filter.

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 (`boolean` is non-portable);
- wrappers (`Boolean` is non-portable);
- `BigDecimal`;
- `BigInteger`;
- `String`;
- `Date`

followed by one of the following words: “`ascending`”, “`descending`”,“`asc`”, or “`desc`”.

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.

Ordering of boolean fields, if supported by the implementation, is `false` before `true`, unless descending is specified. Ordering of null-valued Boolean fields is as above.

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.

14.6.8 Limiting the Cardinality of the Query Result

The application may want to skip some number of results that may have been previously returned, and additionally may want to limit the number of instances returned from a query. The parameters are modeled after `String.getChars` and are 0-origin. The parameters are not saved if the query is serialized. The default range for query execution if this method is not called are `(0, Long.MAX_VALUE)`.

```
setRange(long fromIncl, long toExcl);
```

The `fromIncl` parameter is the number of instances of the query result to skip over before returning the `List` to the user. If specified as 0 (the default), no instances are skipped.

The `toExcl` parameter is the last instance of the query result (before skipping) to return to the user.

The expression `(toExcl - fromIncl)` is the maximum number of instances in the query result to be returned to the user. If fewer instances are available, then fewer instances will be returned. If `((toExcl - fromIncl)<= 0)` evaluates to `true`,

- if the result of the query execution is a `List`, the returned `List` contains no instances, and an `Iterator` obtained from the `List` returns `false` to `hasNext()`.
- if the result of the query execution is a single instance (`setUnique(true)`), it will have a value of `null`.

```
setRange(String range);
```

When using the string form of `setRange` both parameter values are specified either as numbers or as parameters. The `fromIncl` and `toExcl` values are comma separated and evaluated as either long values or as parameter names (beginning with “:”). For example, `setRange(“:fromRange, :toRange”)` or `setRange(“100, 130”)`.

14.6.9 Specifying the Result of a Query (Projections, Aggregates)

The application might want to get results from a query that are not instances of the candidate class. The results might be single-valued fields of persistent instances, instances of classes other than the candidate class, or aggregates of single-valued fields.

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

The result parameter consists of the optional keyword `distinct` followed by a comma-separated list of named result expressions or a constructor expression.

A constructor expression consists of the keyword `new` followed by the name of a result class and a comma-separated parenthesis-enclosed list of named result expressions. See 14.6.12 for a detailed description of the constructor expression.

Distinct results

If `distinct` is specified, the query result does not include any duplicates. If the result parameter specifies more than one result expression, duplicates are those with matching values for each result expression.

Queries against an extent always consider only distinct candidate instances, regardless of whether `distinct` is specified. Queries against a collection might contain duplicate candidate instances; the `distinct` keyword removes duplicates from the candidate collection in this case.

Result expressions begin with either an instance of the candidate class (with an explicit or implicit "this") or an instance of a variable (using the variable name). The candidate tuples are the cartesian product of the candidate class and all variables used in the result. The result tuples are the tuples of the candidate class and all variables used in the result that satisfy the filter. The result is the collection of result expressions projected from the result tuples. If variables are not used in the result expression, then the filter is evaluated for all possible values for each such variable, and if the filter evaluates to true for any combination of such variables, then the candidate tuple becomes a result tuple.

The `distinct` specification requires removing duplicates from projected expressions.

If any result is a navigational expression, and a non-terminal field or variable has a null value for a particular set of conditions (the result calculation would throw `NullPointerException`), then the result is null for that result expression. This is known in relational algebra as “outer join semantics”. For example, to exclude results of “this.department.category.name” where either department or category is null, the user must explicitly add a clause to the filter: “this.department != null && this.department.category != null”.

The result expressions include:

- “`this`”: indicates that the candidate instance is returned
- <field>: this indicates that a field is returned as a value; the field might be in the candidate class or in a class referenced by a variable
- <variable>: this indicates that a variable’s value is returned as a persistent instance
- <aggregate>: this indicates that an aggregate of multiple values is returned; if `null` values are aggregated, they do not participate in the aggregate result; if all of the expressions to be aggregated evaluate to `null`, the result is the same as if there were no instances that match the filter.
  - `count`(<expression>): the count of the number of instances of the expression is returned; the expression is preceded by an optional “distinct” and can be “`this`”, a navigational expression that terminates in a single-valued field, or a variable name
  - `sum`(<numeric field expression>): the sum of field expressions is returned; the expression is preceded by an optional "distinct"
  - `min`(<orderable field expression>): the minimum value of the field expression is returned
  - `max`(<orderable field expression>): the maximum value of the field expression is returned
  - `avg`(<numeric field expression>): the average value of all field expressions is returned; the expression is preceded by an optional "distinct"
- <field expression>: the value of an expression using any of the operators allowed in queries applied to fields is returned

- <navigational expression>: this indicates a navigational path through single-valued fields or variables as specified by the Java language syntax; the navigational path starts with the keyword “this”, a variable, a parameter, or a field name followed by field names separated by dots.
- <parameter>: one of the parameters provided to the query.

The result expression can be explicitly cast using the (cast) operator.

Named Result Expressions

<result expression> as <name>: identify the <result expression> (any of the result expressions specified above) as a named element for the purpose of matching a method or field name in the result class.

If the name is not specified explicitly, the default for name is the expression itself.

Aggregate Types

`Count` returns `Long`.

`Sum` returns `Long` for integral types and the field’s type for other `Number` types (`BigDecimal`, `BigInteger`, `Float`, and `Double`). `Sum` and `avg` are invalid if applied to non-`Number` types.

`Avg`, `min`, and `max` return the type of the expression.

If there are no instances that match the filter,

- `count` returns 0;
- `avg`, `sum`, `min`, and `max` return `null`.

If `null` values are aggregated, they do not participate in the aggregate result. If all of the expressions to be aggregated evaluate to `null`, the result is the same as if there were no instances that match the filter.

Primitive Types

If a result expression has a primitive type, its value is returned as an instance of the corresponding java wrapper class.

Null Results

If the returned value from a query specifying a result is `null`, this indicates that the expression specified as the result was `null`. Note that the semantics of this result are different from the returned value where no instances satisfied the filter.

Default Result

If not specified, the result defaults to “`distinct this as C`” where `C` is the unqualified name of the candidate class. For example, the default result specification for a query where the candidate class is com.acme.hr.Employee is “`distinct this as Employee`”.

14.6.10 Grouping Aggregate Results

Aggregates are most useful if they can be grouped based on an element of the result. Grouping is required if there are aggregate expressions in the result.

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

The grouping parameter consists of one or more expressions separated by commas followed by an optional “having” followed by one Boolean expression.

When grouping is specified, each result expression must be one of:

- an expression contained in the grouping expression; or,
- an aggregate expression evaluated once per group.

When grouping is specified with ordering, each ordering expression must be one of:

- an expression contained in the grouping expression; or,
- an aggregate expression evaluated once per group.

The query groups all elements where all expressions specified in `setGrouping` have the same values. The query result consists of one element per group.

When “`having`” is specified, the “`having`” expression consists of arithmetic and boolean expressions containing expressions that are either aggregate expressions or contained in a grouping expression.

14.6.11 Specifying Uniqueness of the Query Result

If the application knows that there can be exactly zero or one instance returned from a query, the result of the query is most conveniently returned as an instance (possibly `null`) instead of a `List`.

```
void setUnique(boolean unique);
```

When the value of the `Unique` flag is `true`, then the result of a query is a single value, with `null` used to indicate that none of the instances in the candidates satisfied the filter. If more than one instance satisfies the filter, and the range is not limited to one result, then `execute` throws a `JDOUserException`.

Default Unique setting

The default Unique setting is true for aggregate results without a grouping expression, and false otherwise.

14.6.12 Specifying the Class of the Result

The application may have a user-defined class that best represents the results of a query. In this case, the application can specify that instances of this class should be returned.

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

The default result class is the candidate class if the parameter to `setResult` is `null` or not specified. When the result is specified and not `null`, the default result class is the type of the expression if the result consists of one expression, or `Object[]` if the result consists of more than one expression.

Result Class Requirements

- The result class may be one of the `java.lang` classes `Character`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`, or `Object[]`; or one of the `java.math` classes `BigInteger` or `BigDecimal`; or the `java.util` class `Date`; or the `java.util` interface `Map`; or one of the `java.sql` classes `Date`, `Time`, or `Timestamp`; or a user-defined class.
- If there are multiple result expressions, the result class must be able to hold all elements of the result specification or a `JDOUserException` is thrown.
- If there is only one result expression, the result class must be assignable from the type of the result expression or must be able to hold all elements of the result specification. A single value must be able to be coerced into the specified result

class (treating wrapper classes as equivalent to their unwrapped primitive types) or by matching. If the result class does not satisfy these conditions, a `JDOUserException` is thrown.

- A constructor of a result class specified in the constructor expression of the `setResult` method or in the `setResultClass` method will be used if the results specification matches the parameters of the constructor by position and type. If more than one constructor satisfies the requirements, the JDO implementation chooses one of them. If no constructor satisfies the results requirements, the following requirements apply:
  - A user-defined result class must have a no-args constructor and one or more public "set" or "put" methods or fields.
  - Each result expression must match one of:
    - *a public field that matches the name of the result expression and is of the type (treating wrapper types the same as primitive types) of the result expression;*
    - *or if no public field matches the name and type, a public "set" method that returns* `void` *and matches the name of the result expression and takes a single parameter which is the exact type of the result expression;*
    - *or if neither of the above applies,a public method must be found with the signature void put(Object, Object) in which the first argument is the name of the result expression and the second argument is the value from the query result.*
  - Portable result classes do not invoke any persistence behavior during their no-args constructor or "set" methods.

Table 6: Shape of Result (C is the candidate class)

setResult	setResultClass	setUnique	shape of result
null, or "distinct this as C"	null	false	`List<C>`
null, or "distinct this as C"	null	true	C
not null, one result expression of type T	null	false	`List<T>`
not null, one result expression of type T	null	true	T
not null, more than one result expression	null	false	`List<Object[]>`
not null, more than one result expression	null	true	`Object[]`
null or not null	UserResult.class	false	`List<UserResult>`
null or not null	UserResult.class	true	`UserResult`

14.6.13 Single-string Query element binding

The String version of Query represents all query elements using a single string. The string contains the following structure:

select [**unique**] [<result>] [**into** <result-class-name>]

[**from** <candidate-class-name> [**exclude subclasses**]]

[**where** <filter>]

[**variables** <variables-clause>]

[**parameters** <parameters-clause>]

[<imports-clause>]

[**group by** <grouping-clause>]

[**order by** <ordering-clause>]

[**range** <from-range> **,**<to-range>]

Keywords, identified above in **bold**, are either all upper-case or all lower-case. Keywords cannot be mixed case.

The select clause must be the first clause in the query.

The order of the other clauses must be as described above.

If implicit parameters are used, their order of appearance in the query determines their order for binding to positional parameters for execution.

<result> is the result as in 14.6.9.

<result-class-name> is the name of the result class as in 14.6.12.

<filter> is the filter as in 14.6.2.

<variables-clause> is the variable declaration as in 14.6.5. As in Java, variables in the clause are separated by semicolons.

<parameters-clause> is the parameter declaration as in 14.6.3. As in Java, parameters in the clause are separated by commas.

<imports-clause> is the imports declaration as in 14.6.4. As in Java, imports in the clause are separated by semicolons.

<grouping-clause> is the grouping specification as in 14.6.10.

<ordering-clause> is the ordering specification as in 14.6.6.

<from-range> and <to-range> are as in 14.6.8.

14.7 SQL Queries

If the developer knows that the underlying datasource supports SQL, and knows the mapping from the JDO domain model to the SQL schema, it might be convenient in some cases to execute SQL instead of expressing the query as JDOQL. In this case, the factory method that takes the language string and Object is used: `newQuery (String language, Object query)`. The language parameter is “javax.jdo.query.SQL” and the query parameter is the SQL query string.

The SQL query string must be well-formed. The JDO implementation must not make any changes to the query string. The tokens “?” must be used to identify parameters in the SQL query string.

When this factory method is used, the behavior of the `Query` instance changes significantly. The only methods that can be used are `setClass` to establish the candidate class, `setUnique` to declare that there is only one result row, and `setResultClass` to establish the result class.

- there is no filter, and the `setFilter` method throws `JDOUserException`.
- there is no ordering specification, and the `setOrdering` method throws `JDOUserException`.

- there are no variables, and the `declareVariables` method throws `JDOUserException`.
- the parameters are untyped, and the `declareParameters` method throws `JDOUserException`.
- there is no grouping specification, and the `setGrouping` method throws `JDOUserException`.
- the candidate collection can only be the `Extent` of instances of the candidate class, including subclasses, and the `setCandidates` method throws `JDOUserException`.
- parameters are bound by position. If the parameter list is an `Object[]` then the first element in the array is bound to the first "?" in the SQL statement, and so forth. If the parameter list is a `Map`, then the keys of the `Map` must be instances of `Integer` whose `intValue` is 1..n. The value in the `Map` corresponding to the key whose `intValue` is 1 is bound to the first "?" in the SQL statement, and so forth.
- there are no imports, and the `declareImports` method throws `JDOUserException`.
- for queries in which the candidate class is specified, the columns selected in the SQL statement must at least contain the primary key columns of the mapped candidate class, and additionally the discriminator column if defined and the version column(s) if defined.
- results are taken from the SELECT clause of the query, and the `setResult` method throws `JDOUserException`.
- the cardinality of the result is determined by the SQL query itself, and the `setRange` method throws `JDOUserException`.

SQL queries can be defined without a candidate class. These queries can be found by name using the factory method `newNamedQuery`, specifying the class as `null`, or can be constructed without a candidate class.

Table 7: Shape of Result of SQL Query

Candidate class	Selected columns	setResultClass	setUnique	shape of result
C	must include primary key columns	null	false	`List`<C>
C	must include primary key columns	null	true	C
null	single column of type T	null	false	`List`<T>
null	single column of type T	null	true	T
null	more than one result column	null	false	`List`<Object[ ]>
null	more than one result column	null	true	Object[ ]

Table 7: Shape of Result of SQL Query

Candidate class	Selected columns	setResultClass	setUnique	shape of result
null or not null		UserResult.class	false	`List`<UserResult>
null or not null		UserResult.class	true	UserResult

14.7.1 Mapping Columns of SQL Queries to User-specified Result Classes

There are two specified means by which columns of SQL queries can be mapped to user-specified result classes: by name and by position.

Each labeled column in the result set is mapped according to the mapping defined in Section 14.6.12, using the result set column name as the public field or property name of the result class. Since SQL is generally case-insensitive, matching of labels to field and property names is not case-sensitive. Labels that differ only in case cause a `JDOUserException` to be thrown when the query is executed.

A result set column is considered labeled if:

- the return value from `java.sql.ResultSetMetaData.getColumnLabel (int oneBasedColumnIndex)` is non-null and of non-zero length; or,
- if `getColumnLabel` is `null` or of zero length, `java.sql.ResultSetMetaData.getColumnName (int oneBasedColumnIndex)` is non-null and of non-zero length.

Other determinations of whether a column is considered labeled are unspecified and not portable. Each character in a column label that is not a valid character in a Java field or method identifier is converted to an underscore character for the purposes of mapping; other name conversion strategies are not specified and not portable.

Each unlabeled column in the result set is mapped positionally. As required by Section 14.6.12, the result class must expose a public method with the signature `void put(Object, Object)` for data that do not have a public field or set method; it is this method that is used by the implementation to map columns positionally, using the integral position of the column, as an Integer, as the first argument, and the column's value as the second. Positional indexes passed to the result class's `void put(Object, Object)` method are zero-based; that is, the value of the Integer given is one less than the SQL column index, as SQL column indexes are one-based.

14.8 Deletion by Query

An application may want to delete a number of instances in the datastore without instantiating them in memory. The instances to be deleted can be described by a query.

```
long deletePersistentAll(Object[] parameters);
```

```
long deletePersistentAll(Map parameters);
```

```
long deletePersistentAll();
```

These methods delete the instances of affected classes that pass the filter, and all dependent instances. Affected classes are the candidate class and its persistence-capable subclasses. The number of instances of affected classes that were deleted is returned. Embedded instances and dependent instances are not counted in the return value.

Query elements `filter`, `parameters`, `imports`, `variables`, and `unique` are valid in queries used for delete. Elements `result`, `result class`, `range`, `grouping`, and `ordering` are invalid. If any of these elements is set to its non-default value when one of the `deletePersistentAll` methods is called, a `JDOUserException` is thrown and no instances are deleted.

When the value of the `Unique` flag is true, then at most one instance will be deleted. If more than one instance satisfies the filter, then `deletePersistentAll` throws a `JDOUserException`.

Dirty instances of affected classes are first flushed to the datastore. Instances already in the cache when deleted via these methods or brought into the cache as a result of these methods undergo the life cycle transitions as if `deletePersistent` had been called on them.

That is, if an affected class implements the `DeleteCallback` interface, the instances of that class to be deleted are instantiated in memory and the `jdoPreDelete` method is called prior to deleting the instance in the datastore. If any `LifecycleListener` instances are registered with affected classes, these listeners are called for each deleted instance.

Before returning control to the application, instances of affected classes in the cache are refreshed by the implementation so their status in the cache reflects whether they were deleted from the datastore.

14.9 Extensions

Some JDO vendors provide extensions to the query, and these extensions must be set in the query instance prior to execution.

```
void setExtensions(Map extensions);
```

This method replaces all current extensions with the extensions contained as entries in the parameter `Map`. A parameter value of `null` means to remove all extensions. The keys are immediately evaluated; entries where the key refers to a different vendor are ignored; entries where the key prefix matches this vendor but where the full key is unrecognized cause a `JDOUserException` to be thrown. The extensions become part of the state of the `Query` instance that is serialized. The parameter `Map` is not used after the method returns.

```
void addExtension(String key, Object value);
```

This method adds one extension to the `Query` instance. This extension will remain until the next `setExtensions` method is called, or `addExtension` with an equal key. Key recognition behavior is identical to `setExtensions`.

14.10 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;
}
```

14.10.1 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.

```
Query q = pm.newQuery (Employee.class, “salary > 30000”);
Collection emps = (Collection) q.execute ();
<query name=”basic”>
[!CDATA[
select where salary > 30000
]]
</query>
```

14.10.2 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.

```
Query q = pm.newQuery (Employee.class, “salary > 30000”);
q.setOrdering (“salary ascending”);
Collection emps = (Collection) q.execute ();
<query name=”ordering”>
[!CDATA[
select where salary > 30000
order by salary ascending
]]
</query>
```

14.10.3 Parameter passing.

This query selects all `Employee` instances from the candidate collection where the salary is greater than the value passed as a parameter and the name starts with the value passed as a second 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.

```
Query q = pm.newQuery (Employee.class,
   “salary > sal && name.startsWith(begin”);
q.declareParameters (“Float sal, String begin”);
```

```
Collection emps = (Collection) q.execute (new Float (30000.));
<query name=”parameter”>
[!CDATA[
select where salary > :sal && name.startsWith(:begin)
]]
</query>
```

14.10.4 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.

```
Query q = pm.newQuery (Employee.class, “dept.name == dep”);
q.declareParameters (“String dep”);
String rnd = “R&D”;
Collection emps = (Collection) q.execute (rnd);
<query name=”navigate”>
[!CDATA[
select where dept.name == :dep
]]
</query>
```

14.10.5 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.

```
String filter = “emps.contains (emp) & emp.salary > sal”;
Query q = pm.newQuery (Department.class, filter);
q.declareParameters (“float sal”);
q.declareVariables (“Employee emp”);
Collection deps = (Collection) q.execute (new Float (30000.));
<query name=”multivalue”>
[!CDATA[
select where emps.contains(e)
&& e.salary > :sal
]]
</query>
```

14.10.6 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.

```
String filter = “depts.contains(name)”;
Query q = pm.newQuery (Department.class, filter);
List depts =
   Arrays.asList(new String [] {“R&D”, “Sales”, “Marketing”});
q.declareParameters (“Collection depts”);
Collection deps = (Collection) q.execute (depts);
<query name=”collection”>
[!CDATA[
select where :depts.contains(name)
]]
</query>
```

14.10.7 Projection of a Single Field

This query selects names of all `Employee`s who work in the parameter department.

```
Query q = pm.newQuery (Employee.class, “dept.name == deptName”);
q.declareParameters (“String deptName”);
q.setResult(“name”);
Collection names = (Collection) q.execute(“R&D”);
Iterator it = names.iterator();
while (it.hasNext()) {
   String name = (String) it.next();
   ...
}
<query name=”project”>
[!CDATA[
select name where dept.name == :deptName
]]
</query>
```

14.10.8 Projection of Multiple Fields and Expressions

This query selects names, salaries, and bosses of Employees who work in the parameter department.

```
class Info {
   public String name;
   public Float salary;
   public Employee reportsTo;
}
Query q = pm.newQuery (Employee.class, “dept.name == deptName”);
q.declareParameters (“String deptName”);
q.setResult(“name, salary, boss as reportsTo”);
```

```
q.setResultClass(Info.class);
Collection names = (Collection) q.execute(“R&D”);
Iterator it = names.iterator();
while (it.hasNext()) {
   Info info = (Info) it.next();
   String name = info.name;
   Employee boss = info.reportsTo;
   ...
}
<query name=”resultclass”>
[!CDATA[
select name, salary, boss as reportsTo into Info
where dept.name == :deptName
]]
</query>
```

14.10.9 Projection of Multiple Fields and Expressions into a Constructed instance

This query selects names, salaries, and bosses of Employees who work in the parameter department, and uses the constructor for the result class.

```
class Info {
   public String name;
   public Float salary;
   public Employee reportsTo;
   public Info (String name, Float salary, Employee reportsTo) {
      this.name = name;
      this.salary = salary;
      this.reportsTo = reportsTo;
   }
}
Query q = pm.newQuery (Employee.class, “dept.name == deptName”);
q.declareParameters (“String deptName”);
q.setResult(“new Info(name, salary, boss)”);
q.setResultClass(Info.class);
Collection names = (Collection) q.execute(“R&D”);
Iterator it = names.iterator();
while (it.hasNext()) {
   Info info = (Info) it.next();
   String name = info.name;
   Employee boss = info.reportsTo;
   ...
}
<query name=”construct”>
[!CDATA[
select new Info (name, salary, boss)
where dept.name == :deptName
]]
</query>
```

14.10.10 Aggregation of a single Field

This query averages the salaries of `Employee`s who work in the parameter department and returns a single value.

```
Query q = pm.newQuery (Employee.class, “dept.name == deptName”);
q.declareParameters (“String deptName”);
q.setResult(“avg(salary)”);
Float avgSalary = (Float) q.execute(“R&D”);
<query name=”aggregate”>
[!CDATA[
select avg(salary)
where dept.name == :deptName
]]
</query>
```

14.10.11 Aggregation of Multiple Fields and Expressions

This query averages and sums the salaries of `Employee`s who work in the parameter department.

```
Query q = pm.newQuery (Employee.class, “dept.name == deptName”);
q.declareParameters (“String deptName”);
q.setResult(“avg(salary), sum(salary)”);
Object[] avgSum = Object[] q.execute(“R&D”);
Float average = (Float)avgSum[0];
Float sum = (Float)avgSum[1];
<query name=”multiple”>
[!CDATA[
select avg(salary), sum(salary)
where dept.name == :deptName
]]
</query>
```

14.10.12 Aggregation of Multiple fields with Grouping

This query averages and sums the salaries of `Employee`s who work in all departments having more than one employee and aggregates by department name.

```
Query q = pm.newQuery (Employee.class);
q.setResult(“avg(salary), sum(salary), dept.name”);
q.setGrouping(“dept.name having count(dept.name) > 1”);
Collection results = (Collection)q.execute();
Iterator it = results.iterator();
while (it.hasNext()) {
   Object[] info = (Object[]) it.next();
   Float average = (Float)info[0];
   Float sum = (Float)info[1];
   String deptName = (String)info[2];
   ...
}
<query name=”group”>
[!CDATA[
select avg(salary), sum(salary), dept.name from com.xyz.hr.Employee where dept.name == :deptName group by dept.name having count(dept.name) > 1
]]
</query>
```

14.10.13 Selection of a Single Instance

This query returns a single instance of `Employee`.

```
Query q = pm.newQuery (Employee.class, “name == empName”);
q.declareParameters (“String empName”);
q.setUnique(true);
Employee emp = (Employee) q.execute(“Michael”);
<query name=”unique”>
[!CDATA[
select unique this
where dept.name == :deptName
]]
</query>
```

14.10.14 Selection of a Single Field

This query returns a single field of a single `Employee`.

```
Query q = pm.newQuery (Employee.class, “name == empName”);
```

```
q.declareParameters (“String empName”);
q.setResult(“salary”);
q.setResultClass(Float.class);
q.setUnique(true);
Float salary = (Float) q.execute (“Michael”);
<query name=”single”>
[!CDATA[
select unique new Float(salary)
where dept.name == :deptName
]]
</query>
```

14.10.15 Projection of “this” to User-defined Result Class with Matching Field

This query selects instances of `Employee` who make more than the parameter salary and stores the result in a user-defined class. Since the default is “distinct this as Employee”, the field must be named Employee and be of type Employee.

```
class EmpWrapper {
   public Employee Employee;
}
Query q = pm.newQuery (Employee.class, “salary > sal”);
q.declareParameters (“Float sal”);
q.setResultClass(EmpWrapper.class);
Collection infos = (Collection) q.execute (new Float (30000.));
Iterator it = infos.iterator();
while (it.hasNext()) {
   EmpWrapper info = (EmpWrapper)it.next();
   Employee e = info.Employee;
   ...
}
<query name=”thisfield”>
[!CDATA[
select into EmpWrapper
where salary > sal
]]
</query>
```

14.10.16 Projection of “this” to User-defined Result Class with Matching Method

This query selects instances of `Employee` who make more than the parameter salary and stores the result in a user-defined class.

```
class EmpInfo {
    private Employee worker;
    public Employee getWorker() {return worker;}
    public void setEmployee(Employee e) {worker = e;}
}
Query q = pm.newQuery (Employee.class, “salary > sal”);
q.declareParameters (“Float sal”);
q.setResultClass(EmpInfo.class);
Collection infos = (Collection) q.execute (new Float (30000.));
Iterator it = infos.iterator();
while (it.hasNext()) {
    EmpInfo info = (EmpInfo)it.next();
    Employee e = info.getWorker();
    ...
}

<query name=”thismethod”>
[!CDATA[
select into EmpInfo
where salary > sal
]]
</query>
```

14.10.17 Projection of variables

This query returns the names of all `Employees` of all "Research" departments:

```
Query q = pm.newQuery(Department.class);
q.declareVariables("Employee e");
q.setFilter("name.startsWith('Research') && emps.contains(e)");
q.setResult(e.name);
Collection names = q.execute();
Iterator it = names.iterator();
while (it.hasNext()) {
    String name = (String)it.next();
    ...
}
<query name=”variables”>
[!CDATA[
select e.name
where name.startsWith('Research')
&& emps.contains((com.xyz.hr.Employee) e)
]]
</query>
```

14.10.18 Deleting Multiple Instances

This query deletes all `Employee`s who make more than the parameter salary.

```
Query q = pm.newQuery (Employee.class, “salary > sal”);
q.declareParameters (“Float sal”);
q.deletePersistentAll(new Float(30000.));
```

15 Object-Relational Mapping

JDO is datastore-independent. However, many JDO implementations support storage of persistent instances in relational databases, and this storage requires that the domain object model be mapped to the relational schema. The mapping strategies for simple cases are for the most part the same from one JDO implementation to another. For example, typically a class is mapped to one or more tables, and fields are mapped to one or more columns.

The most common mapping paradigms are standardized, which allows users to define their mapping once and use the mapping for multiple implementations.

Mapping Overview

Mapping between the domain object model and the relational database schema is specified from the perspective of the object model. Each class is mapped to a primary table and possibly multiple secondary tables and multiple join tables. Fields in the class are mapped to columns in either the primary table, secondary tables, or join tables. Simple field types typically map to single columns. Complex field types (`Collection`s, `Map`s, and `array`s) typically map to multiple columns.

Secondary tables represent non-normalized tables that contain zero or one row corresponding to each row in the primary table, and contain field values for the persistent class. These tables might be modeled as one-to-one relationships, but they can be modeled as containing nullable field values instead.

Secondary tables might be used by a single field mapping or by multiple field mappings. If used by a single field mapping, the join conditions linking the primary and secondary table might be specified in the field mapping itself. If used by multiple field mappings, the join conditions might be specified in each field mapping or specified in the class mapping.

Complex field types are mapped by mapping each of the components individually. Collections map the element and optional order components. Maps map the key and value components. Arrays map the element and order components.

Mapping Strategies

The specification does not standardize how the mapping files are generated. Most implementations will support one or more of the following strategies for creating mapping files:

- starting with a relational schema, generate persistence-capable classes and the mapping to relate them (sometimes referred to as reverse mapping or class generation);
- starting with persistence-capable classes, generate the relational schema and the mapping to relate them (sometimes called forward mapping or schema generation);
- starting with a relational schema and persistence-capable classes, create the mapping to relate them (sometimes called meet-in-the-middle mapping).

This specification does not standardize how the mapping files are created. Implementations might support command-line or interactive GUI-based tools to assist in the process.

There is no portable behavior for incompletely specified mappings. When a portable application runs, the mapping is completely specified by the mapping metadata, regardless of whether the user created the mapping or the mapping was created by a tool. If the mapping is incompletely specifed, the JDO implementation might silently use mapping defaults or throw an exception.

15.1 Column Elements

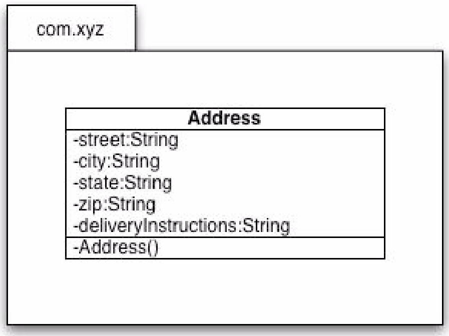
Column elements used for simple, non-relationship field value mapping specify at least the column name. The field value is loaded from the value of the named column.

The column element might contain additional information about the column, for use in generating schema. This might include the scale and precision for numeric types, the maximum length for variable-length field types, the jdbc type of the column, or the sql type of the column. This information is ignored for runtime use, with the following exception: if the jdbc type of the column does not match the default jdbc type for the field's class (for example, a String field is mapped to a CLOB rather than a VARCHAR column), the jdbc type information is required at runtime.

Column elements that contain only the column name can be omitted, if the column name is instead contained in the enclosing element. Thus, a field element is defined to allow a column attribute if only the name is needed, or a column element if more than the name is needed. If both column attribute and column element are specified for any element, it is a user error.

15.1.1 Mapping single-valued fields to columns

This example demonstrates mappings between fields and value columns.



```
package com.xyz;
public class Address {
   String street;
   String city;
   String state;
   String zip;
   String deliveryInstructions;
}
```

```
CREATE TABLE ADDR (
    STREET VARCHAR(255) PRIMARY KEY,
    CITY VARCHAR(255),
    STATE CHAR(2),
    ZIPCODE VARCHAR(10),
    DELIV_INS CLOB
)

<orm>
    <package name="com.xyz">
        <class name="Address" table="ADDR">
            <field name="street" column="STREET"/>
            <field name="city" column="CITY"/>
            <field name="state" column="STATE"/>
            <field name="zip" column="ZIPCODE"/>
            <field name="deliveryInstructions">
               <column name="DELIV_INS" jdbc-type="CLOB"/>
            </field>
        </class>
    </package>
</orm>
```

15.2 Join Condition

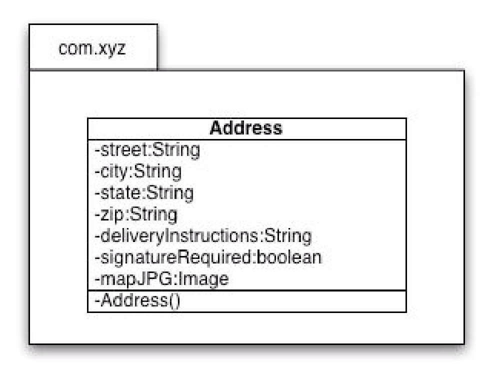
Secondary tables and join tables are mapped using a join condition that associates a column or columns in the secondary or join table with a column or columns in the primary table, typically the primary table’s primary key columns.

Column elements used for relationship mapping or join conditions specify the column name and optionally the target column name. The target column name is the name of the column in the associated table corresponding to the named column. The target column name is optional when the target column is the single primary key column of the associated table, or when the target column name is identical to the join column name.

NOTE: ***This usage of column elements is fundamentally different from the usage of column elements for value mapping. For value mapping, the name attribute names the column that contains the value to be used. For join conditions, the name attribute names the column that contains the reference data to be joined to the primary key column of the target.***

15.2.1 Secondary Table mapping

This example demonstrates the use of `join` elements to represent join conditions linking a class' primary table and secondary tables used by fields.

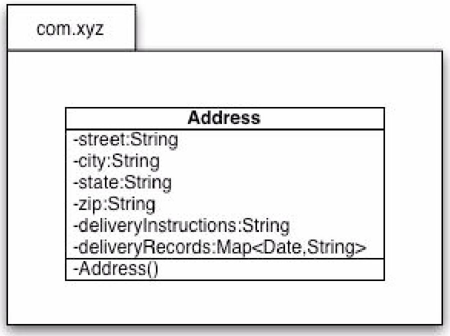


```
package com.xyz;
public class Address {
   String street;
   String city;
   String state;
   String zip;
   String deliveryInstructions;
   boolean signatureRequired;
   byte[] mapJPG;
}

CREATE TABLE ADDR (
    STREET VARCHAR(255) PRIMARY KEY,
    CITY VARCHAR(255),
    STATE CHAR(2),
    ZIPCODE VARCHAR(10)
)

CREATE TABLE DELIV (
    ADDR_STREET VARCHAR(255),
    SIG_REQUIRED BIT,
    DELIV_INS CLOB
)

CREATE TABLE MAPQUEST_INFO (
    ADDR_STREET VARCHAR(255),
    MAPQUEST_IMAGE BLOB
)
```

```
<orm>
   <package name="com.xyz">
      <class name="Address" table="ADDR">
         <!-- shared join condition used by fields in DELIV -->
         <join table="DELIV" column="ADDR_STREET"/>
         <field name="street" column="STREET"/>
         <field name="city" column="CITY"/>
         <field name="state" column="STATE"/>
         <field name="zip" column="ZIPCODE"/>
         <field name="signatureRequired" table="DELIV"
            column="SIG_REQUIRED"/>
         <field name="deliveryInstructions" table="DELIV">
            <column name="DELIV_INS" jdbc-type="CLOB"/>
         </field>
         <field name="mapJPG" table="MAPQUEST_INFO"
            column="MAPQUEST_IMAGE">
         <!-- join condition defined for this field only -->
            <join column="ADDR_STREET"/>
         </field>
      </class>
   </package>
</orm>
```

15.2.2 Map using join table

This example uses the <join> element to map a `Map<Date,String>` field to a join table. Note that in this example, the primary table has a compound primary key, requiring the use of the target attribute in join conditions.



```
package com.xyz;
public class Address {
   String street;
   String city;
   String state;
   String zip;
   String deliveryInstructions;
   boolean signatureRequired;
   Map<Date,String> deliveryRecords;
}

CREATE TABLE ADDR (
    STREET VARCHAR(255),
    CITY VARCHAR(255),
    STATE CHAR(2),
    ZIPCODE VARCHAR(10),
    PRIMARY KEY (STREET, ZIPCODE)
)

CREATE TABLE DELIV_RECORDS (
    ADDR_STREET VARCHAR(255),
    ADDR_ZIPCODE VARCHAR(10),
    DELIV_DATE TIMESTAMP,
    SIGNED_BY VARCHAR(255)
)

<orm>
   <package name="com.xyz">
      <class name="Address" table="ADDR">
         <field name="street" column="STREET"/>
         <field name="city" column="CITY"/>
         <field name=”state” column=”STATE”/>
         <field name="zip" column="ZIPCODE"/>
         <!-- field type is Map<Date,String> -->
         <field name="deliveryRecords" table="DELIV_RECORDS">
            <join>
               <column name="ADDR_STREET" target="STREET"/>
               <column name="ADDR_ZIPCODE" target="ZIPCODE"/>
            </join>
            <key column="DELIV_DATE"/>
            <value column="SIGNED_BY"/>
         </field>
      </class>
   </package>
</orm>
```

15.3 Relationship Mapping

Column elements used for relationship mapping are contained in either the field element directly in the case of a simple reference, or in one of the collection, map, or array elements contained in the field element.

In case only the column name is needed for mapping, the column name might be specified in the field, collection, or array element directly instead of requiring a column element with only a name.

The field on the other side of the relationship can be mapped simply by identifying the field on the other side that defines the mapping, using the mapped-by attribute. Changes to the field mapped via “mapped-by” are not reflected in the datastore. There is no further relationship implied by having both sides of the relationship map to the same database column(s). In particular, making a change to one side of the relationship does not imply any runtime behavior by the JDO implementation to change the other side of the relationship in memory, although the column(s) will be changed during commit and will therefore be visible by both sides in the next transaction.

If two relationships (one on each side of an association) are mapped to the same column, the field on only one side of the association needs to be explicitly mapped.

The field on the other side of the relationship can be mapped by using the mapped-by attribute identifying the field on the side that defines the mapping. Regardless of which side changes the relationship, flush (whether done as part of commit or explicitly by the user) will modify the datastore to reflect the change and will update the memory model for consistency. There is no further behavior implied by having both sides of the relationship map to the same database column(s). In particular, making a change to one side of the relationship does not imply any runtime behavior by the JDO implementation to change the other side of the relationship in memory prior to flush, and there is no requirement to load fields affected by the change if they are not already loaded. This implies that if the `RetainValues` flag or `DetachAllOnCommit` is set to `true`, and the relationship field is loaded, then the implementation will change the field on the other side so it is visible after transaction completion.

Conflicting changes to relationships cause a `JDOUserException` to be thrown at flush time. Conflicting changes include:

- adding a related instance with a single-valued mapped-by relationship field to more than one one-to-many collection relationship
- setting both sides of a one-to-one relationship such that they do not refer to each other

Mapping Strategies

For single-valued relationships, there are three basic ways to map references from one persistence-capable class (the referring class) to a related class:

- serialized: The entire related instance is serialized into a single column in the primary or secondary table of the referring class.

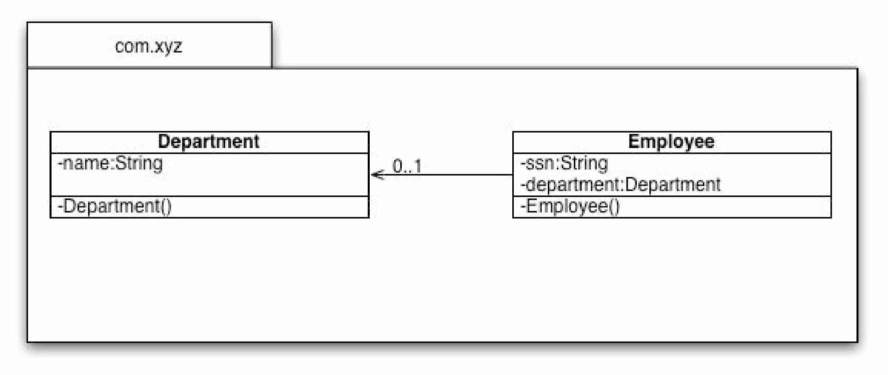
- embedded: The related instance is mapped, field by field, to columns in the primary or secondary table of the referring class.
- by reference: The related instance is in a different table, and the column in the primary or secondary table of the referring class contains a reference (often, a foreign key) to the primary table of the related class.

For multi-valued relationships, there are five basic ways to map references from one persistence-capable class (the referring class) to a related class:

- serialized: The entire collection, array, or map is serialized into a single column in the primary or secondary table of the referring class.
- serialized in a join table: A join table is used to associate multiple rows in the join table with a single row in the primary or secondary table of the referring class, and the related instances are serialized, one per row, into a single column in the join table.
- embedded in a join table: A join table is used to associate multiple rows in the join table with a single row in the primary or secondary table of the referring class, and each related instance is mapped, one per row, field by field, into multiple columns in the join table.
- by reference to the primary table of the related class: The related class has a reference (often, a foreign key) to the primary table of the referring class.
- by reference in a join table: A join table is used to associate multiple rows in the join table with a single row in the primary or secondary table of the referring class, and a column in the join table contains a reference (often, a foreign key) to the primary table of the related class.

15.3.1 Many-to-One using foreign key

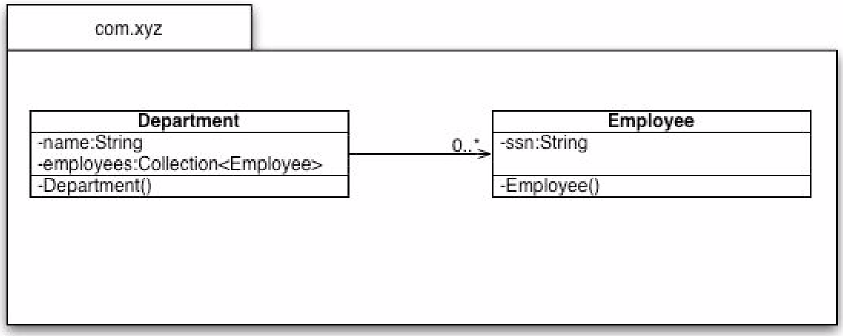
A many-one mapping (Employee has a reference to Department).



```
package com.xyz;
public class Department {
   String name;
}

public class Employee {
   String ssn;
   Department department;
}

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    DEP_NAME VARCHAR(255)
)
CREATE TABLE DEP (
    NAME VARCHAR(255) PRIMARY KEY
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <!-- field type is Department -->
         <field name="department" column="DEP_NAME"/>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
      </class>
   </package>
</orm>
```

15.3.2 One-to-Many using foreign key

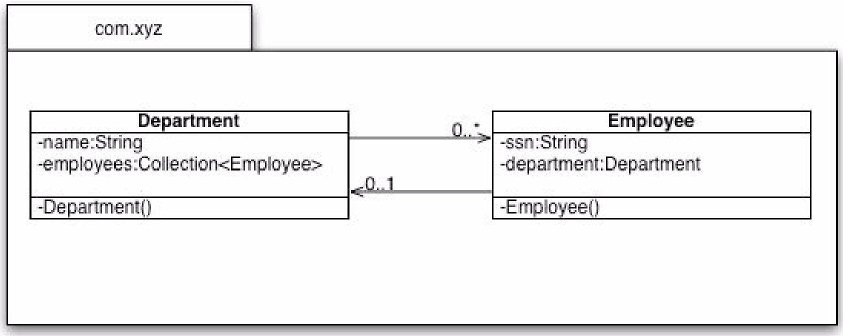
A one-many mapping (Department has a collection of Employees). This example uses the same schema as Example 4.



```
package com.xyz;
public class Department {
   String name;
   Collection<Employee> employees;
}

public class Employee {
   String ssn;
}

<orm>
   <package name="com.xyz">
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <!-- field type is Collection<Employee> -->
         <field name="employees">
            <element column="DEP_NAME"/>
         </field>
      </class>
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
      </class>
   </package>
</orm>
```

15.3.3 Many-to-One and One-to-Many using mapped-by

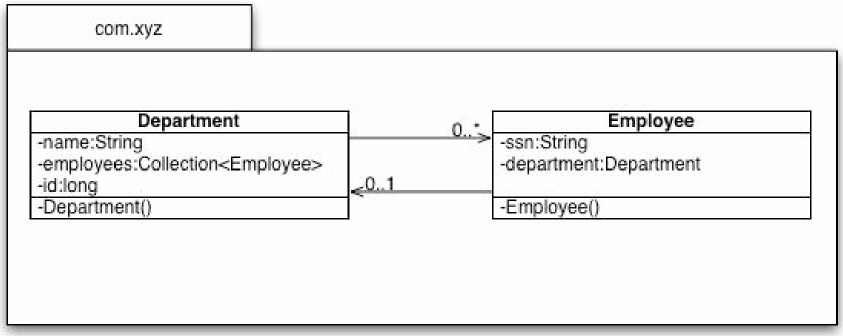
If both the Employee.department and Department.employees fields exist, only one needs to be mapped explicitly; one side is specified to be “mapped-by” the other side. The Department side is marked as using the same mapping as a field on the Employee side. This example uses the same schema as Examples 4 and 5.



```
package com.xyz;
public class Department {
   String name;
   Collection<Employee> employees;
}

public class Employee {
   String ssn;
   Department department;
}

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <field name="department" column="DEP_NAME"/>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <field name="employees" mapped-by="department"/>
      </class>
   </package>
</orm>
```

15.3.4 Many-to-One and One-to-Many using compound foreign key

This example mirrors Example 6, but now Department has a compound primary key.

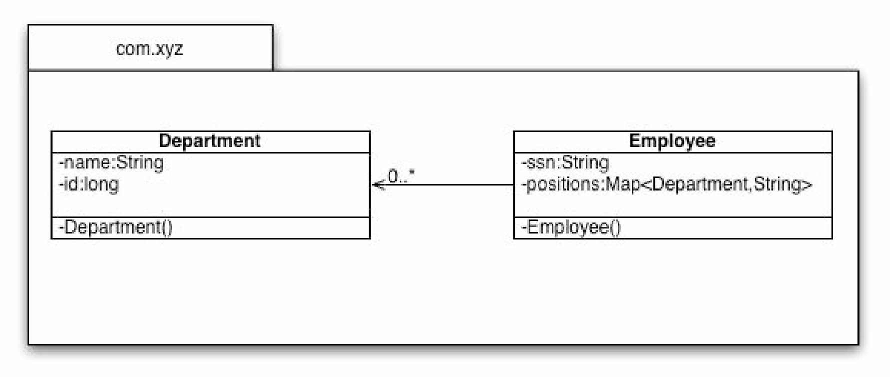


```
package com.xyz;
public class Department {
   String name;
   Collection<Employee> employees;
   long id;
}

public class Employee {
   String ssn;
   Department department;
}

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    DEP_NAME VARCHAR(255),
    DEP_ID BIGINT
)

CREATE TABLE DEP (
    NAME VARCHAR(255),
    ID BIGINT,
    PRIMARY KEY (NAME, DEP_ID)
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <field name="department">
            <column name="DEP_NAME" target="NAME"/>
            <column name="DEP_ID" target="ID"/>
         </field>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <field name="id" column="ID"/>
         <field name="employees" mapped-by="department"/>
      </class>
   </package>
</orm>
```

15.3.5 Many-to-One and One-to-Many using Map<Department, String>

Employee has a `Map<Department, String>` mapping each department the employee is a member of to her position within that department. Department still has a compound primary key.

The `Map` uses a join table that contains one row for each entry in the `Map`. The columns in the join table refer to the `Employee`, the `Department`, and the position.



```
package com.xyz;
public class Department {
   String name;
   long id;
}

public class Employee {
   String ssn;
   Map<Department,String> positions;
}

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY
)

CREATE TABLE DEP (
    NAME VARCHAR(255),
    ID BIGINT,
    PRIMARY KEY (NAME, ID)
)

CREATE TABLE EMP_POS (
    EMP_SSN CHAR(10),
    DEP_NAME VARCHAR(255)
    DEP_ID BIGINT,
    POS VARCHAR(255)
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <!-- field type is Map<Department, String> -->
         <field name="positions" table="EMP_POS">
            <join column="EMP_SSN"/>
            <key>
               <column name="DEP_NAME" target="NAME"/>
               <column name="DEP_ID" target="ID"/>
            </key>
            <value column="POS"/>
         </field>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <field name="id" column="ID"/>
      </class>
   </package>
</orm>
```

15.3.6 Many-to-One and One-to-Many using Map<String, Employee>

`Department` has a `Map<String, Employee>` mapping the role in the department to the employee. Department still has a compound primary key.

The `Map` uses the employee's table that contains the role as well as other employee information. The mapping on the `Department` side uses the mapped-by attribute naming the field in the `Employee` that refers to `Department`. The `key` uses the `mapped-by` attribute naming the field in `Employee` that contains the key for the map.

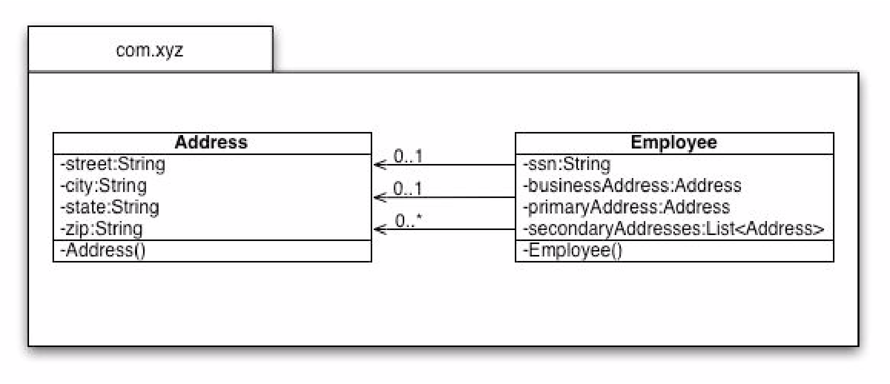
```
package com.xyz;
public class Department {
   String name;
   long id;
   Map<String, Employee> roles;
}

public class Employee {
   String ssn;
   Department dept;
   String role;
}

CREATE TABLE EMP (
   SSN CHAR(10) PRIMARY KEY,
   DEPT BIGINT,
   ROLE VARCHAR
)

CREATE TABLE DEP (
    NAME VARCHAR(255),
    ID BIGINT,
    PRIMARY KEY (NAME, ID)
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <field name=”dept” column=”DEP”/>
         <field name=”role” column=”ROLE”/>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <field name="id" column="ID"/>
         <!-- field type is Map<String, Employee> -->
         <field name="roles" mapped-by=”dept”>
            <key mapped-by=”role”/>
         </field>
      </class>
   </package>
</orm>
```

15.4 Embedding

Some of the columns in a table might be mapped as a separate Java class to better match the object model. Embedding works to arbitrary depth.

15.4.1 Mapping relationships using embedded, referenced, and join table

Employee has a reference to a business address, which is a standard many-one. Employee also has a primary Address, whose data is embedded within the Employee record. Finally, Employee has a List<Address> of secondary Address references, whose data is embedded in the join table.



```
package com.xyz;
public class Address {
   String street;
   String city;
   String state;
   String zip;
}

public class Employee {
   String ssn;
   Address businessAddress;
   Address primaryAddress;
   List<Address> secondaryAddresses;
}

CREATE TABLE ADDR (
    STREET VARCHAR(255) PRIMARY KEY,
    CITY VARCHAR(255),
    STATE CHAR(2),
    ZIPCODE VARCHAR(10)
)

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    BUSADDR_STREET VARCHAR(255),
    PADDR_STREET VARCHAR(255),
    PADDR_CITY VARCHAR(255),
    PADDR_STATE CHAR(2),
    PADDR_ZIPCODE VARCHAR(10)
)

CREATE TABLE EMP_ADDRS (
    EMP_SSN CHAR(10),
    IDX INTEGER,
    SADDR_STREET VARCHAR(255),
    SADDR_CITY VARCHAR(255),
    SADDR_STATE CHAR(2),
    SADDR_ZIPCODE VARCHAR(10)
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <!-- field type is Address -->
         <field name="businessAddress" column="BUSADDR_STREET"/>
         <!-- field type is Address -->
         <field name="primaryAddress">
            <embedded null-indicator-column="PADDR_STREET">
               <field name="street" column="PADDR_STREET"/>
               <field name="city" column="PADDR_CITY"/>
               <field name="state" column="PADDR_STATE"/>
               <field name="zip" column="PADDR_ZIPCODE"/>
            </embedded>
         </field>
         <!-- field type is List<Address> -->
         <field name="secondaryAddresses" table="EMP_ADDRS">
            <join column="EMP_SSN"/>
            <element>
               <embedded>
                  <field name="street" column="SADDR_STREET"/>
                  <field name="city" column="SADDR_CITY"/>
                  <field name="state" column="SADDR_STATE"/>
```

```
                    <field name="zip" column="SADDR_ZIPCODE"/>
                </embedded>
            </element>
            <order column="IDX"/>
        </field>
    </class>
  </package>
</orm>
```

15.5 Foreign Key Constraints

Foreign keys in metadata serve two quite different purposes. First, when generating schema, the foreign key element identifies foreign keys to be generated. Second, when using the database, foreign key elements identify foreign keys that are assumed to exist in the database. This is important for the runtime to properly order insert, update, and delete statements to avoid constraint violations. Foreign keys are part of ORM metadata and are probably meaningless in non-relational implementations.

Foreign key constraints can be generated in three ways:

- Most elements that can include nested `column` elements can define `delete-action` or `update-action` attributes.
- Most elements that can contain nested `column` elements can define a nested `foreign-key` element. This element has the following attributes:
  - name: the name of the generated constraint
  - deferred: boolean attribute describing whether the constraint evaluation is deferred until datastore commit
  - delete-action: the foreign key delete action; see below. In this case, the "none" value is not allowed.
  - update-action: the foreign key update action; see below.
- The `class` element can define `foreign-key` elements. A class-level `foreign-key` element has the `name`, `deferred`, `delete-action`, `and update-action` attributes as above.

Note that regardless of which side of a relationship in the object model is mapped, the meaning of delete action and update action refer to the columns in the datastore, not to the fields in the object model.

Delete Action, Update Action

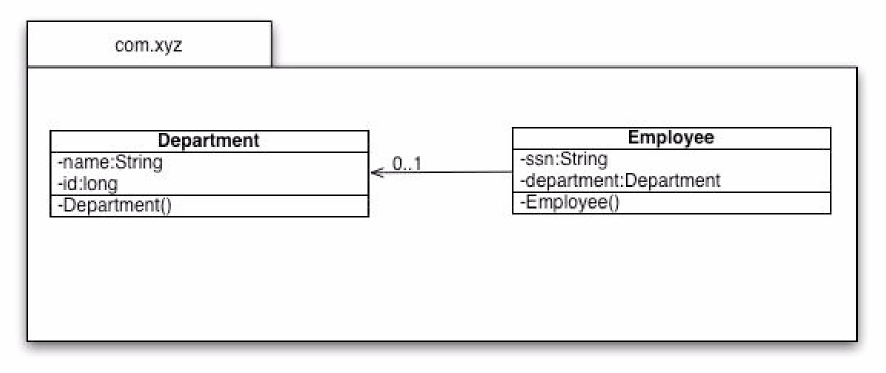
The delete-action and update-action attributes have the following permitted values:

- “none”: no foreign key is generated and none is assumed to exist; no special action is required of the implementation
- “restrict” (the default): a foreign key with the “restrict” delete action is generated or is assumed to exist; the implementation will require update and delete statements to be executed in proper sequence
- “cascade”: a foreign key with the “cascade” delete action is generated or is assumed to exist; the database will automatically delete all rows that refer to the row being deleted

- "null": a foreign key with the "null" delete action is generated or is assumed to exist; a referring key will be nullified if the target key is updated or deleted
- "default": a foreign key with the "default" delete action is generated or is assumed to exist

15.5.1 Many-to-One with foreign key constraint

A many-one relation from Employee to Department, represented by a standard restrict-action database foreign key.



```
package com.xyz;
public class Department {
   String name;
   long id;
}

public class Employee {
   String ssn;
   Department department;
}

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    DEP_NAME VARCHAR(255),
    DEP_ID BIGINT,
   FOREIGN KEY EMP_DEP_FK (DEP_NAME, DEP_ID) REFERENCES DEP (NAME, ID)
)

CREATE TABLE DEP (
    NAME VARCHAR(255),
    ID BIGINT,
    PRIMARY KEY (NAME, DEP_ID)
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <field name="ssn" column="SSN"/>
         <field name="department">
            <column name="DEP_NAME" target="NAME"/>
            <column name="DEP_ID" target="ID"/>
            <foreign-key name="EMP_DEP_FK"/>
         </field>
      </class>
      <class name="Department" table="DEP">
         <field name="name" column="NAME"/>
         <field name="id" column="ID"/>
      </class>
   </package>
</orm>
```

15.6 Indexes

Index definitions are used for schema generation and are not used at runtime. In relational implementations, they are part of the ORM metadata because their names and extensions might differ for each database. In non-relational implementations, indexes are part of the JDO metadata.

Indexes can be defined in three ways:

- Most elements that can include nested `column` elements can define an `indexed` attribute. This attribute has three possible values:
  - true: generate a standard index on the datastore representation of the element
  - false: do not generate an index on the element
  - unique: generate a unique index on the element
- Most elements that can contain nested `column` elements can define a nested `index` element. The element does not contain any elements (aside from possible extensions). The index is generated on the datastore representation of the parent element. This element has the following attributes:
  - name: the name of the generated index
  - unique: boolean attribute describing whether to generate a unique index

- The `class` element can define nested `index` elements. A class-level `index` element has the attributes outlined above. It can contain `column` and/or `field` elements, each of which is limited to a `name` attribute referencing a column or field defined elsewhere. Field names can use `<superclass-name>.<field-name>` syntax to reference superclass fields, `<field-name>.<embedded-field-name>` to reference embedded relation fields, and the `#key`, `#value`, and `#element` suffixes defined for fetch groups to reference parts of a field.

Unique Constraints

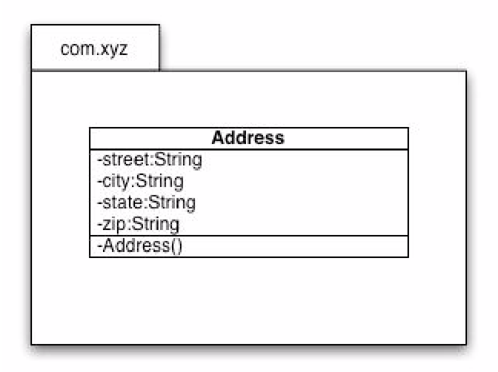
Unique constraints are used during schema generation, and may be used at runtime to order datastore operations. Like indexes, they are part of ORM metadata in relational implementations, and part of JDO metadata in non-relational implementations.

Unique constraints can be defined in the same three general ways as indexes:

- Most elements that can include nested `column` elements can define an `unique` attribute. Possible values are `true` and `false`.
- Most elements that can contain nested column elements can define a nested `unique` element. This element has the following attributes:
  - name: the name of the generated constraint
  - deferred: boolean attribute describing whether the constraint evaluation is deferred until datastore commit
- The `class` element can contain `unique` elements. A class-level `unique` element has the attributes outlined above. It contains the same possible elements as a class-level index.

15.6.1 Single-field and Compound Indexes

This example demonstrates single-field and compound indexes.



```
package com.xyz;
public class Address {
   String street;
   String city;
   String state;
   String zip;
}
```

```
CREATE TABLE ADDR (
    STREET VARCHAR(255) PRIMARY KEY,
    CITY VARCHAR(255),
    STATE CHAR(2),
    ZIPCODE VARCHAR(10)
)

<orm>
   <package name="com.xyz">
      <class name="Address" table="ADDR">
         <field name="street" column="STREET"/>
         <field name="city" column="CITY"/>
         <field name="state" column="STATE"/>
         <field name="zip" column="ZIPCODE">
            <index name="ADDR_ZIP_IDX"/>
         </field>
         <index name="ADDR_CITYSTATE_IDX">
            <column name="CITY"/>
            <column name="STATE"/>
         </index>
      </class>
   </package>
</orm>
```

15.7 Inheritance

Each class can declare an inheritance strategy. Three strategies are supported by standard metadata: new-table, superclass-table, and subclass-table.

- new-table creates a new table for the fields of the class.
- superclass-table maps the fields of the class into the superclass table.
- subclass-table forces subclasses to map the fields of the class to their own table.

Using these strategies, standard metadata directly supports several common inheritance patterns, as well as combinations of these patterns within a single inheritance hierarchy.

One common pattern uses one table for an entire inheritance hierarchy. A column called the discriminator column is used to determine to which class each row belongs. This pattern is achieved by a strategy of new-table for the base class, and superclass-table for all subclasses. These are the default strategies for base classes and subclasses when no explicit strategy is given.

Another pattern uses multiple tables joined by their primary keys. In this pattern, the existence of a row in a table determines the class of the row. A discriminator column is not required, but may be used to increase the efficiency of certain operations. This pattern is achieved by a strategy of new-table for the base class, and new-table for all subclasses. In this case, the join element specifies the columns to be used for associating the columns in the table mapped by the subclass(es) and the table mapped by the superclass.

A third pattern maps fields of superclasses and subclasses into subclass tables. This pattern is achieved by a strategy of subclass-table for the base class, and new-table for direct subclasses.

15.8 Versioning

Three common strategies for versioning instances are supported by standard metadata. These include state-comparison, timestamp, and version-number.

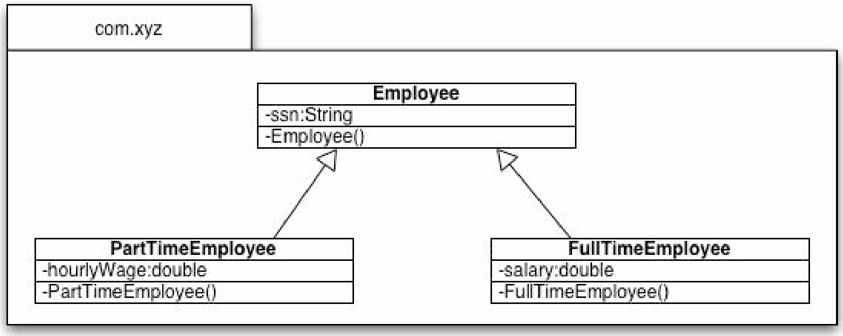
State-comparison involves comparing the values in specific columns to determine if the database row was changed.

Timestamp involves comparing the value in a date-time column in the table. The first time in a transaction the row is updated, the timestamp value is updated to the current time.

Version-number involves comparing the value in a numeric column in the table. The first time in a transaction the row is updated, the version-number column value is incremented.

15.8.1 Inheritance with superclass-table and version

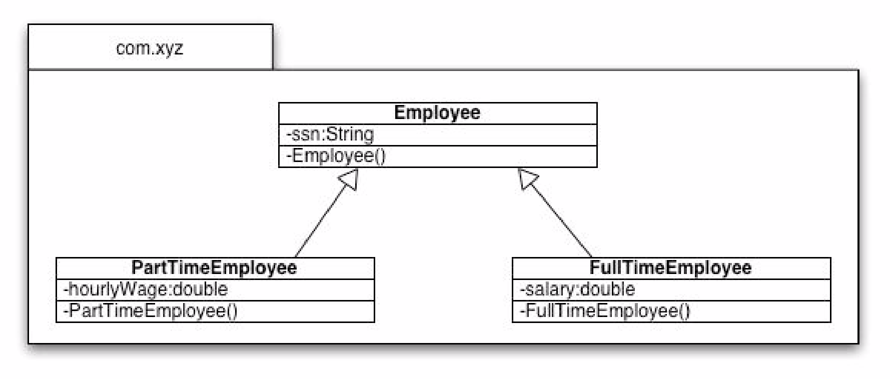
Mapping a subclass to the base class table, and using version-number optimistic versioning. Note that in this example, the inheritance strategy attribute is not needed, because this is the default inheritance pattern. The version strategy attribute is also using the default value, and could have been omitted. These attributes are included for clarity.



```
package com.xyz;
public class Employee {
   String ssn;
}
public class PartTimeEmployee extends Employee {
   double hourlyWage;
}
public class FullTimeEmployee extends Employee {
   double salary;
}

CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    TYPE CHAR(1),
    WAGE FLOAT,
    SALARY FLOAT,
    VERS INTEGER
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <inheritance strategy="new-table">
            <discriminator value="E" column="TYPE"/>
         </inheritance>
         <field name="ssn" column="SSN"/>
         <version strategy="version-number" column="VERS"/>
      </class>
      <class name="PartTimeEmployee">
         <inheritance strategy="superclass-table">
            <discriminator value="P"/>
         </inheritance>
         <field name="hourlyWage" column="WAGE"/>
      </class>
      <class name="FullTimeEmployee">
         <inheritance strategy="superclass-table">
            <discriminator value="F"/>
         </inheritance>
         <field name="salary" column="SALARY"/>
      </class>
   </package>
</orm>
```

15.8.2 Inheritance with new-table and version

Mapping each class to its own table, and using state-image versioning. Though a discriminator is not required for this inheritance pattern, this mapping chooses to use one to make some actions more efficient. It stores the full Java class name in each row of the base table.

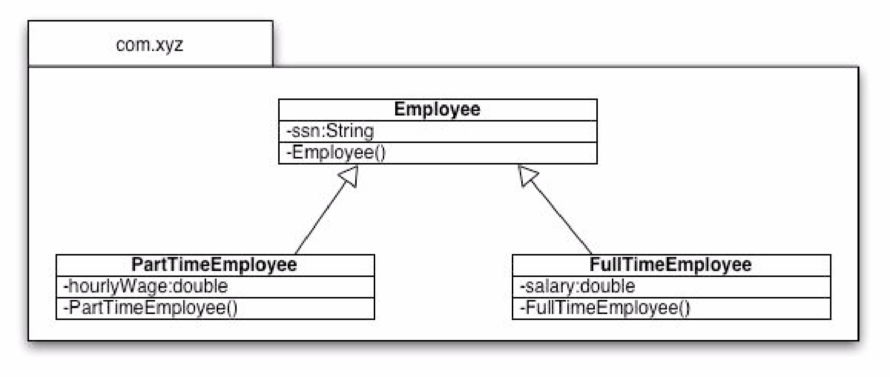


```
CREATE TABLE EMP (
    SSN CHAR(10) PRIMARY KEY,
    JAVA_CLS VARCHAR(255)
)

CREATE TABLE  PART_EMP (
    EMP_SSN CHAR(10) PRIMARY KEY,
    WAGE FLOAT
)

CREATE TABLE FULL_EMP (
    EMP_SSN CHAR(10) PRIMARY KEY,
    SALARY FLOAT
)

<orm>
   <package name="com.xyz">
      <class name="Employee" table="EMP">
         <inheritance strategy="new-table">
            <discriminator strategy="class-name" column="JAVA_CLS"/>
         </inheritance>
         <field name="ssn" column="SSN"/>
         <version strategy="state-comparison"/>
      </class>
      <class name="PartTimeEmployee" table="PART_EMP>
         <inheritance strategy="new-table">
            <join column="EMP_SSN"/>
         </inheritance>
         <field name="hourlyWage" column="WAGE"/>
```

```
        </class>
        <class name="FullTimeEmployee" table="FULL_EMP">
            <inheritance strategy="new-table">
                <join column="EMP_SSN"/>
            </inheritance>
            <field name="salary" column="SALARY"/>
        </class>
    </package>
</orm>
```

15.8.3 Inheritance with subclass-table

This example maps superclass fields to each subclass table.



```
CREATE TABLE  PART_EMP (
    EMP_SSN CHAR(10) PRIMARY KEY,
    WAGE FLOAT
)

CREATE TABLE FULL_EMP (
    EMP_SSN CHAR(10) PRIMARY KEY,
    SALARY FLOAT
)

<orm>
    <package name="com.xyz">
        <class name="Employee">
            <inheritance strategy="subclass-table"/>
        </class>
        <class name="PartTimeEmployee" table="PART_EMP”>
            <inheritance strategy="new-table"/>
            <field name="Employee.ssn" column="EMP_SSN"/>
            <field name="hourlyWage" column="WAGE"/>
         </class>
         <class name="FullTimeEmployee" table="FULL_EMP">
            <inheritance strategy="new-table"/>
            <field name="Employee.ssn" column="EMP_SSN"/>
            <field name="salary" column="SALARY"/>
         </class>
      </package>
   </orm>
```

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`. *How to test?(JDO must know JTA..)* 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` *How to test?* and automatically commit the `UserTransaction` during `javax.jdo.Transaction.commit`. *How to test?*

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

While it is possible for container-managed persistence entity beans to be implemented by the container using JDO, the implementation details are beyond the scope of this document.

It is possible for users to implement bean-managed persistence entity beans using JDO, but implementation details are container-specific and no recommendations for the general case are given.

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 persistence-capable. 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.

17.1.11 JDODetachedFieldAccessException

This exception is the result of a user accessing a field of a detached instance, where the field was not copied to the detached instance. It is a derived class of `JDOUserException`.

18 XML Metadata

This chapter specifies the metadata that describes a persistence-capable class, optionally including its mapping to a relational database. The metadata is stored in XML format. For implementations that support binary compatibility, 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.

NOTE: J2SE 5 introduced standard elements for defining the types of collections and maps. Because of these features, programs compiled with suitable type information might not need a separate file to describe type information.

Metadata annotations for persistence are being developed in JSR 220. When that specification is final, an update to the JDO specification to specify support for the annotations will be made.

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, classes, and mappings, regardless of their physical packaging.

The metadata associated with each persistence capable class must be contained within one or more files, and its format is defined by the DTD or xsd. If the metadata in a file 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.

Metadata for all classes and interfaces found while processing metadata for any class or interface must be remembered by the implementation.

Metadata for relational mapping might be contained in the same file as the persistence information, in which case the naming convention above is used. The mapping metadata might be contained in a separate file, in which case the metadata file name suffix must be specified in the `PersistenceManagerFactory` property `javax.jdo.option.Mapping`. This property is used to construct the file names for the mapping.

NOTE: If the `javax.jdo.option.Mapping` property is set, then mapping metadata contained in the .jdo file **is not used**.

The extension .orm refers to “object repository metadata”. If the mapping is to a repository type other than relational, the document type will be different, but the file naming conventions are the same.

For example, if the value of `javax.jdo.option.Mapping` is “mySQL”, then the file name for the metadata is <class-name>-mySQL.orm or package-mySQL.orm. Similar to package.jdo, the package-mySQL.orm file is located in one of the following directories: “META-INF”; “WEB-INF”; <none>, in which case the metadata file name is "package-mySQL.orm" with no directory; “<package>/.../<package>”, in which case the metadata directory name is the partial or full package name with “package-mySQL.orm” as the file name. If mapping metadata is for only one class, the name of the file is <package>/.../<package>/<class-name>-mySQL.orm.

When metadata information is needed for a class, and the metadata for that class has not already been loaded, the metadata is searched for 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 as long as the class is not garbage collected. Therefore, metadata contained higher in the search order will always be used instead of metadata contained lower in the search order.

Similarly, when mapping metadata information is needed for a class, and the mapping metadata for that class has not already been loaded, the mapping metadata is searched for as follows: META-INF/package-mySQL.orm, WEB-INF/package-mySQL.orm, package-mySQL.orm, <package>/.../<package>/package-mySQL.orm, and <package>/.../<package>/<class-name>-mySQL.orm. Once mapping metadata for a class has been loaded, it will not be replaced as long as the class is not garbage collected. Therefore, mapping 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.Grumpy. In this case, subsequent search of metadata for org.acme.Grumpy 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 or property declarations are missing from the metadata, then field or property metadata is defaulted for the missing declarations. The JDO implementation is able to determine based on the metadata whether a class is persistence-capable or not. 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.

Classes and interfaces used in metadata follow the Java rules for naming. If the class or interface name is unqualified, the package name is the name of the enclosing package. Inner classes are identified by the `“$”` marker.

For compatibility with installed applications, a JDO implementation might first use the search order as specified in the JDO 1.0 or 1.0.1 releases. In this case, if metadata is not found, then the search order as specified in JDO 2.0 must be used. Refer to Chapter 25 for details.

For convenience, the metadata allows for the same information to be declared in multiple places. It is an error if conflicting information is declared in more than one place. For example, the name of the column for a field might be declared either in the column attribute on the field element, or in the name attribute in the column element contained in the field. If declared in both places, the information must be identical or an error must be reported by the JDO implementation.

Mapping to Relational Databases

Mapping is done by specifying associations from classes and interfaces to tables, and fields to columns.

Tables are generally specified by name. Table names can be declared as "<database>.<catalog>.<schema>.<table-name>", where database, catalog, and schema are optional. If not specified in any metadata, catalog and schema are taken from the `PersistenceManagerFactory` properties `catalog` and `schema`. If not specified in `PersistenceManagerFactory`, they are defaulted by the JDBC connection.

Catalog and schema attributes apply to jdo, orm, package, class, and interface elements, and specify the catalog and schema to be used when defining and using schema. If declared at the jdo, orm, package, class or interface level, it specifies the catalog and/or schema to be used as the default for tables contained therein.

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. It contains multiple package and query elements and optional extension elements.

18.2 ELEMENT package

This element includes all classes in a particular package. The complete qualified package name defaults to the empty package, but it is highly recommended to specify it. It contains multiple class and interface elements and optional extension elements.

18.3 ELEMENT interface

The `interface` element declares a persistence-capable interface. Instances of a vendor-specific type that implement this interface can be created using the `newInstance(Class persistenceCapable)` method in `PersistenceManager`, and these instances may be made persistent.

The JDO implementation must maintain an extent for persistent instances of persistence-capable classes that implement this interface.

The `requires-extent` attribute is optional. If set to `“false”`, the JDO implementation does not need to support extents of factory-made persistent instances. It defaults to `“true”`.

The attribute `name` is required, and is the name of the interface.

The attribute `table` is optional, and is the name of the table to be used to store persistent instances of this interface.

The `detachable` attribute specifies whether persistent instances of this interface can be detached from the persistence context and later attached to the same or a different persistence context. The default is `false`.

Persistent fields declared in the interface are defined as those that have both a `get` and a `set` method or both an `is` and a `set` method, named according to the JavaBeans naming conventions, and of a type supported as a persistent type.

The implementing class will provide a suitable implementation for all property access methods and will throw `JDOUserException` for all other methods of the interface.

This element might contain `property` elements to specify the mapping to relational columns.

Interface inheritance is supported.

18.4 ELEMENT column

The `column` element identifies a column in a mapped table. This element is used for mapping fields, collection elements, array elements, keys, values, datastore identity, application identity, and properties.

NOTE: Any time an element can contain a `column` element that is only used to name the column, a `column` attribute can be used instead.

The `name` attribute declares the name of the column in the database. The name might be fully qualified as <table-name>.<column-name> and <table-name> might be defaulted in context.

The `target` attribute declares the name of the primary key column for the referenced table. For columns contained in join elements, this is the name of the primary key column in the primary table. For columns contained in field, element, key, value, or array elements, the `target` attribute is the name of the primary key column of the primary table of the other side of the relationship.

The `target-field` attribute might be used instead of the `target` attribute to declare the name of the field to which the column refers. This is useful in cases where there are different mappings of the referenced field in different subclasses.

The `jdbc-type` attribute is used to determine the type of the column in the database. This type is defaulted based on the type of the field being mapped. Valid types are all upper-case or all lower-case CHAR, VARCHAR, LONGVARCHAR, NUMERIC, DECIMAL, BIT, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, and TIMESTAMP, and others as may be defined by future versions of the JDBC specification. This attribute is only needed if the default type is not suitable.

The `jdbc-type` is also used when mapping `element`, `key`, `value`, and `order` elements of collections, arrays, and maps. The java type for the column mapped to an `order` element is assumed to be `int`.

For example, when mapping a `Map<Integer, Employee>` to a join table, the jdbc-type for the column mapped to the key (Integer) will default to INTEGER, whereas there is no default jdbc-type for the column mapped to the value (Employee).

Table 8: Default jdbc-type

Java type	Default jdbc-type	Comments
boolean	BIT	
java.lang.Boolean	BIT	
char	CHAR	
java.lang.Character	CHAR	
byte	TINYINT	
java.lang.Byte	TINYINT	
short	SMALLINT	
java.lang.Short	SMALLINT	
int	INTEGER	
java.lang.Integer	INTEGER	
long	BIGINT	
java.lang.Long	BIGINT	
float	FLOAT	
java.lang.Float	FLOAT	
double	DOUBLE	
java.lang.Double	DOUBLE	
java.util.Date	TIMESTAMP	
java.sql.Date	DATE	
java.sql.Time	TIME	
java.sql.Timestamp	TIMESTAMP	
java.lang.Object	none	
java.lang.String	VARCHAR	
java.util.Locale	VARCHAR	
java.util.Currency	VARCHAR	
java.math.BigInteger	NUMERIC	
java.math.BigDecimal	DECIMAL	
interfaces	none	

Table 8: Default jdbc-type

Java type	Default jdbc-type	Comments
mapped as serialized	LONG VARBINARY	
persistence-capable types	none	

In many cases, the default for the `jdbc-type` attribute based on the field type is sufficient. For cases where this information is used to create datastore schema, the jdo implementation is free to map the column type suitable for the datastore being used based on the specified `jdbc-type` , `length`, and `scale`.

The `sql-type` attribute declares the type of the column in the database. This type is database-specific and should only be used where the user needs more explicit control over the mapping. Normally, the combination of `jdbc-type`. `length`, and `scale` are sufficient for the JDO implementation to calculate the `sql-type`.

The `length` attribute declares the number of characters in the datastore representation of numeric, `char[]`, and `Character[]` types; and the maximum number of characters in the datastore representation of `String` types. The default is 256.

The `scale` attribute declares the scale of the numeric representation in the database. The default is 0.

The `allows-null` attribute specifies whether `null` values are allowed in the column, and is defaulted based on the type of the field being mapped. The default is `"true"` for reference field types and `"false"` for primitive field types.

The `insert-value` attribute specifies the value to be inserted into the datastore in case a column is not mapped to any field in the object model. In this case, the `column` element must be directly contained in a `class` element, and the column must not be mapped to a field.

The `default-value` attribute specifies the database-assigned default value for the column if no value is explicitly assigned to the column on insert. Implementations might use the value of this attribute to set the appropriate column default when generating schema.

18.5 ELEMENT class

The `class` element includes `field` elements declared in a persistence-capable class, and optional vendor extensions.

The `name` attribute of the class is required. It specifies the unqualified class name of the class. The class name is scoped by the name of the package in which the class element is contained.

`The persistence-modifier` attribute specifies whether this class is persistence-capable, persistence-aware, or non-persistent. Persistence-aware and non-persistent classes must not include any attributes or elements except for the `name` and `persistence-modifier` attributes. Declaring persistence-aware and non-persistent classes might provide a performance improvement for enhancement and runtime, as the search algorithm for metdata need not be exhaustive.

The `detachable` attribute specifies whether instances of this class can be detached from the persistence context and later attached to the same or a different persistence context. If a class is declared as detachable, then all its persistence-capable subclasses are also detachable. The default is `false`.

The `embedded-only` attribute declares whether instances of this class are permitted to exist as first-class instances in the datastore. A value of “true” means that instances can only be embedded in other first-class instances., and precludes mapping this class to its own table.

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 any field declares the `primary-key` attribute to be `true`; and `datastore`, if not. If the identity type is application, the object-id class is not specified, and there is one primary key field that matches the type of a single field identity class, then the object-id class defaults to that single field identity 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 deprecated for this release. The attribute will be ignored so metadata files from previous releases can be used.A number of `join` elements might be contained in the class element. Each `join` element defines a table and associated join conditions that can be used by multiple fields in the mapping.

The `objectid-class` attribute identifies the name of the objectid class. If not specified, there must be only one primary key field, and the `objectid-class` defaults to the appropriate simple identity class.

The `objectid-class` attribute is required only for abstract classes and classes with multiple key fields. If the `objectid-class` attribute is defined in any concrete persistence-capable class, then the objectid class itself must be concrete, and no subclass of the persistence-capable 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 table attribute names the primary table to which fields declared in this class metadata are mapped.

Foreign keys, indexes, and join tables can be specified at the class level. If they are specified at this level, column information might only be the names of the columns.

Column elements can be added to the class element to describe columns that are not mapped to fields. In this case, the insert-value attribute can be used to specify the value to insert into the column when a new instance is inserted into the datastore.

18.5.1 ELEMENT datastore-identity

The `datastore-identity` element declares the strategy for implementing datastore identity for the class, including the mapping of the identity columns of the relational table.

The `strategy` attribute identifies the strategy for mapping.

- The value `“native”` allows the JDO implementation to pick the most suitable strategy based on the underlying database.
- The value `“sequence”` specifies that a named database sequence is used to generate key values for the table. If `sequence` is used, then the `sequence` attribute is required.
- The value `“autoassign”` specifies that the column identified as the key column is managed by the database to automatically increment key values.
- The value `“identity”` specifies that the column identified as the key column is managed by the database as an identity type.
- The value `“increment”` specifies a strategy that simply finds the largest key already in the database and increments the key value for new instances. It can be used with integral column types when the JDO application is the only database user inserting new instances.
- The value `“uuid-string”` specifies a strategy that generates a 128-bit UUID unique within a network (the IP address of the machine running the application is part of the id) and represents the result as a 16-character String.
- The value `“uuid-hex”` specifies a strategy that generates a 128-bit UUID unique within a network (the IP address of the machine running the application is part of the id) and represents the result as a 32-character String.

The `sequence` attribute names the sequence used to generate key values. This must correspond to a named sequence in the JDO metadata. If this attribute is used, the strategy defaults to “`sequence`”.

The `column` elements identify the primary key columns for the table in the database.

18.5.2 ELEMENT version

The `version` element is contained in the `class` element, and declares the version strategy and optionally the column(s) used for the version strategy.

The `strategy` attribute defines the strategy for managing the version of an instance. Four `strategy` attribute values are standard:

- `none`: no version checking is done; changed values overwrite values in the datastore
- `version-number`: a rolling number is used as the version number
- `state-image`: the values of fields are used in aggregate as the version
- `date-time`: a clock timestamp (date-plus-time) value is used as the version

The `column` attribute declares the name of the column to hold the version. It is used instead of the contained `column` element in case only the column name is needed.

The `version` element might contain one or more `column` elements that declare the columns to use to hold the `version`.

18.6 ELEMENT primary-key

The `primary-key` element provides the mapping for the primary key constraint for the table associated with the enclosing element (`class`, `join`, or `interface`). Its primary use is to specify the name of the primary key constraint in case of Java-to-database mapping. In this case, the element is typically specified as:

```
<primary-key name="EMP_PK"/>
```

It is also optionally used to specify the column to be used for surrogate primary key with application identity. In this case, the primary key fields do not provide the primary key of the database. This mapping is not required to be supported by the JDO implementation. For example:

```
<class name="Employee" identity-type="application">
  <primary-key name="EMP_PK" column="SURR_PK"/>
...
</class>
```

It is also optionally used to specify the constraint name and column names of a primary key constraint for tables associated with the class, such as join tables or secondary tables. To specify the primary key constraint for a join table, the `primary-key` element is contained within the `join` element. For example:

```
<field name="projects" table="EMP_PROJ">
  <collection element-type="Project"/>
  <join>
    <primary-key name="EMP_PROJ_PK">
      <column name="EMPID"/>
      <column name="PROJID"/>
    </primary-key>
    <column name="EMPID" target="ID"/>
  </join>
  <element>
    <column name="PROJID" target="ID"/>
  </element>
</field>
```

If used to specify the primary key for a subclass using new-table inheritance strategy with a join to the superclass table, the primary-key element is put at the class level and not in the join element of the inheritance element.

18.7 ELEMENT join

The `join` element declares the table to be used in the mapping and the join conditions to associate rows in the joined table to the primary table.

The `table` attribute specifies the name of the table in the case of secondary table mappings (at least one table in addition to the primary table contain columns mapped to fields). In this case, the `join` element is nested in the `class` element.

For `join` elements nested inside `field` elements, the `table` attribute is not allowed. The `table` attribute from the `field` element specifies the table to which the join applies.

One or more `column` elements are contained within the `join` element. The `column` elements name the columns used to join to the primary key columns of the primary table. If there are multiple key columns, then the `target` attribute is required in each `column` element, and each names the corresponding primary key column of the primary table.

The table being joined might not have a row for each row in the referring table; in order to access rows in this table, an outer join is needed. The `outer` attribute indicates that an outer join is needed. The default is `false`.

18.8 ELEMENT inheritance

The `inheritance` element declares the mapping for inheritance.

The `strategy` attribute declares the strategy for mapping:

- The value “`subclass-table`” means that this class does not have its own table. All of its fields are mapped by subclasses.
- The value `“new-table”` means that this class has its own table into which by default all of its fields are mapped. There might be a table attribute specified in the class element. This is the default for the topmost (least derived) class in an inheritance hierarchy.
- The value `“superclass-table”` means that this class does not have its own table. All of its fields by default are mapped into tables of its superclass(es). This is the default for all classes except for the topmost class in an inheritance hierarchy.

18.9 ELEMENT discriminator

The `discriminator` element is used when a column is used to identify what class is associated with the primary key value in a table mapped to a superclass.

In the least-derived class in the hierarchy that uses the discriminator strategy, declare the discriminator element with a strategy and column. If the strategy is `“value-map”`, then for each concrete subclass, define the discriminator element with a value attribute. If the strategy is `“class-name”` then subclasses do not need a discriminator element; the name of the class is stored as the value for the row in the table. If the value attribute is given, then the strategy defaults to “value-map”.

The strategy “none” declares that there is no discriminator column.

18.10 ELEMENT implements

The `implements` element declares a persistence-capable interface implemented by the persistence-capable class that contains this element. An extent of persistence-capable classes that implement this interface is managed by the JDO implementation. The extent can be used for queries or for iteration just like an extent of persistence-capable instances.

The attribute `name` is required, and is the name of the interface. The java class naming rules apply: if the interface name is unqualified, the package is the name of the enclosing package.

18.11 ELEMENT foreign-key

The `foreign-key` element specifies characteristics of a foreign key associated with the containing join, field, key, value, or element.

To specify that there is a foreign key associated with the containing element, without specifying the name of the foreign key, the `foreign-key` element can be used with no attributes or contained elements.

If this element is specified at the class level, then `column` elements contained in the `foreign-key` element might contain only the `name` attribute.

18.11.1 ATTRIBUTE update-action

The `update-action` attribute specifies the generated or assumed foreign key constraint defined in the datastore. The implementation might optimize its behavior based on these constraints but they do not affect the object model. The permitted values `restrict`, `cascade`, `default`, `null`, and `none` correspond to the meaning of these terms in SQL.

18.11.2 ATTRIBUTE delete-action

The `delete-action` attribute specifies the generated or assumed foreign key constraint defined in the datastore. The implementation might optimize its behavior based on these constraints but they do not affect the object model. The permitted values `restrict`, `cascade`, `default`, `null`, and `none` correspond to the meaning of these terms in SQL.

18.11.3 ATTRIBUTE deferred

The deferred attribute specifies whether constraint checking on the containing element is defined in the database as being deferred until commit. This allows an optimization by the JDO implementation, and might allow certain operations to succeed where they would normally fail. For example, to exchange unique references between pairs of objects requires that the unique constraint columns temporarily contain duplicate values.

Possible values are “`true`” and “`false`”. The default is “`false`”.

18.11.4 ATTRIBUTE name

The `name` attribute specifies the name of the foreign key constraint to generate for this mapping. This attribute is used if the name of the foreign key needs to be specified.

18.12 ELEMENT unique

The `unique` element specifies characteristics of a unique key associated with the containing join, field, key, value, or element.

To specify that there is a unique key associated with the containing element, without specifying the name of the unique key, the `unique` element can be used with no attributes or contained elements. Alternatively, the `unique` attribute can be used.

If this element is specified at the class level, then `column` elements contained in the `unique` element might contain only the `name` attribute.

18.13 ELEMENT index

The `index` element specifies characteristics of an index associated with the containing join, field, key, value, or element.

To specify that there is an index associated with the containing element, without specifying the name of the index, the `index` element can be used with no attributes or contained elements. Alternatively, the `indexed` attribute can be used.

If this element is specified at the class level, then `column` elements contained in the `foreign-key` element might contain only the `name` attribute.

18.14 ELEMENT property

When contained in a `class` element,

- the `property` element declares the mapping between a virtual field of an implemented interface and the corresponding persistent field of the persistence-capable class.
- the `name` attribute is required, and declares the name for the property. The naming conventions for JavaBeans property names is used: the property name is the same as the corresponding `get` method for the property with the `get` or `is` removed and the resulting name lower-cased.
- the `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.
- the `field-name` attribute is required; it associates a persistent field with the named property.

When contained in an `interface` element,

- `property` elements declare the mapping for persistent properties of the interface.
- The `name` attribute is required and must match the name of a property in the interface.
- This element might contain `column` elements to specify the mapping to relational columns.
- the `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.
- The element might contain `collection`, `map`, or `array` elements to specify the characteristics of the property.

18.15 ELEMENT field

The `field` element 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`: `Currency`, `Date`, `Locale`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedHashMap`, `LinkedHashSet`, `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 persistence-capable 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 `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 `load-fetch-group` attribute specifies the name of the fetch group to be used when this field is loaded due to being referenced when unloaded. It does not apply to queries, navigation, or `getObjectById` of instances of the declaring class.

- The `load-fetch-group` is added to the fetch groups in the `PersistenceManager`’s `FetchPlan` to create the effective fetch groups for loading the unloaded field. The unloaded field is also added to the fields in the effective fetch groups in case the unloaded field is not already defined in the effective fetch groups.

- The effective fetch groups are used to retrieve unloaded fields into the instance containing the unloaded field.
- If any relationship fields are included in the effective fetch groups, then the referred instances are loaded according to the effective fetch groups.

Embedded

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”` for other types including persistence-capable types, interface types and the `Object` type. Thus, specifying this attribute is not usually necessary. 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.

For relational mapping, the embedded attribute is not used for collection, map, and array types, but is only used for persistence-capable types, interface types, and the `Object` type. For other datastores, it only applies to the structure of the type, not to the elements, keys, and values. That is, the collection instance itself is considered separate from its contents. The contents of these types may separately be specified to be embedded or not, using `embedded-element`, `embedded-key`, and `embedded-value` attributes of the `collection`, `array`, and `map` elements.

`Embedded-element`, `embedded-key`, and `embedded-value` apply to the storage of the element, key, and value instances contained in the collection, array, or map. Similar to the `embedded` attribute, for relational mapping these are only applicable to persistence-capable types, interface types, and the `Object` type. For other datastores, these attributes default to “true” for elements, keys, and values of wrapper types, `java.lang` types, `java.math` types and for the types explicitly mapped using the `embedded` element contained in the `element`, `key`, and `value` elements. Like the `embedded` attribute, usually `embedded-element`, `embedded-key`, and `embedded-value` will default appropriately and need not be specified.

The `embedded` attribute applied to a field of a persistence-capable type specifies that the implementation will treat the field as a Second Class Object.

The `serialized` attribute indicates that the field is to be serialized for storage using `writeObject`, and cannot be queried.

The attributes `serialized=”true”` and `embedded=”true”`are mutually exclusive.

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 `embedded` element is used to specify the field mappings for embedded persistence-capable types.

The `dependent` attribute specifies that the field contains a reference to a referred instance that is to be deleted from the datastore if the instance in which the field is declared is deleted, or if the referring field is nullified.

Dependent defaults to `true` if either `serialized=”true”` or `embedded=”true”` is specified.

The `field-type` attribute is used to specify a more restrictive type than the field definition in the class. This might be required in order to map the field to the datastore. To be portable, specify the name of a single type that is itself able to be mapped to the datastore (e.g. a field of type `Object` can specify `field-type="Integer"`). To specify multiple types that the field might contain, use a comma-separated list of types, although this cannot be portably mapped. Rules for type names are as specified in `collection element-type`.

Column Mapping

Non-relationship fields are mapped to one or more columns in the primary table, a secondary table, or a join table. Relationship fields can additionally be mapped to columns in the primary or secondary table of the associated class. The table attribute in the field element identifies one of the three tables. If not specified, the table attribute defaults to the primary table.

Secondary tables are used for mapping single-valued types (primitive, wrapper, `java.util.Date`, `String`, etc.). There is one row in the secondary table for each row in the primary table. The column or columns to which the field is mapped refers to a column or columns in the secondary table.

Join tables can be used for mapping multi-valued types (collection, array, and map types). There are multiple rows in the join table for each row in the primary table. The column or columns to which the field is mapped refers to a column or columns in the join table.

A portable mapping for arrays, collections, and maps will include a primary key on the join table.

A special case involves self-referencing fields, in which the type of a field is the same as its class (or the element, key, or value type is the same). A column mapped to a self-referencing field is in the primary table of the class, and contains a reference to the primary key of the primary table.

If a join element is specified as part of the field mapping, the join column (or columns) provide the join condition to relate the primary table of the class to the table specified in the field. In this case the table attribute in the join element is not used.

The following field declarations are mutually exclusive; it is a user error to specify more than one mutually exclusive declaration:

- `default-fetch-group = "true"`
- `primary-key = "true"`
- `persistence-modifier = "transactional" or "none"`

If `default-fetch-group` is specified as `true`, then `primary-key` is set to `false` and `persistence-modifier` is set to `persistent`.

If `primary-key` is specified as `true`, then `default-fetch-group` is set to `false` and `persistence-modifier` is set to `persistent`.

If `persistence-modifier` is specified as `transactional` or `none`, `default-fetch-group` is set to `false` and `primary-key` is set to `false`.

The `table` attribute specifies the name of the table mapped to this field. It defaults to the table declared in the enclosing `class` element.

The column elements specify the column(s) mapped to this field. Normally, only one column is mapped to a field. If multiple columns are mapped, then the behavior is implementation-specific.

The `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.

The `value-strategy` attribute specifies the strategy used to generate values for the field. This attribute has the same values and meaning as the `strategy` attribute in datastore-identity.

If the `value-strategy` is `sequence`, the `sequence` attribute specifies the name of the sequence to use to automatically generate a value for the field. This value is used only for persistent-new instances at the time `makePersistent` is called.

Subclasses might map fields of their superclasses. In this case, the field name is specified as <superclass>.<superclass-field-name>.

Foreign key

The `delete-action`, `update-action`, `indexed`, `and unique` attributes specify the characteristics of a constraint to be generated or assumed to exist in the database, corresponding to the mapped column or columns.

18.15.1 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 language 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. Classes `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Number`, `Object`, `Short`, `String`, and `StringBuffer` are treated exactly as in the Java language: they are first checked to see if they are in the package in which they are used, and if not, assumed to be in the `java.lang` package. To be portable, specify the name of a single type that is itself able to be mapped to the datastore (e.g. a field of type `Object` can specify `field-type=”Integer”`). To specify multiple types that the field might contain, use a comma-separated list of types, although this cannot be portably mapped.

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 persistence-capable types, the `Object` type, interface types and multi-valued types; and `"true"` for primitive types, wrapper types, and single-valued `Object` types including `String`, `Date`, and `Locale`. It also defaults to `true` if the `element` element contains the `embedded` element specifying the embedded mapping of the element.

The embedded treatment of the collection instance itself is governed by the `embedded` attribute of the `field` element.

The `dependent-element` attribute indicates that the collection’s element contains a reference to a referred instance that is to be deleted if the referring instance is deleted, the collection is replaced, or the reference is nullified or removed from the collection.

The `element` element contained in the `field` element specifies the mapping of elements in the collection.

The `serialized-element` attribute indicates that the array element is to be serialized for storage using `writeObject`, and cannot be queried.

The attributes `serialized-element=”true”` and `embedded-element=”true”`are mutually exclusive.

18.15.2 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. They follow the same rules as `element-type` in element `collection`.

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 persistence-capable types, the `Object` type, interface types and multi-valued types; and `"true"` for primitive types, wrapper types, and single-valued `Object` types including `String`, `Date`, and `Locale`. They also default to `"true"` if the `key` or `value` elements contain the `embedded` element specifying the embedded mapping of the key or value.

The `serialized-key` attribute indicates that the map key is to be serialized for storage using `writeObject`, and cannot be queried.

The attributes `serialized-key=”true”` and `embedded-key=”true”`are mutually exclusive.

The embedded treatment of the map instance itself is governed by the `embedded` attribute of the `field` element.

The `dependent-key` attribute indicates that the collection’s key contains references that are to be deleted if the referring instance is deleted, the map is replaced, or the key is removed from the map.

The `dependent-value` attribute indicates that the collection’s value contains references that are to be deleted if the referring instance is deleted, the map is replaced, or the value is removed from the map.

The `serialized-value` attribute indicates that the map value is to be serialized for storage using `writeObject`, and cannot be queried.

The attributes `serialized-value=”true”` and `embedded-value=”true”`are mutually exclusive.

18.15.3 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 section 18.10.

The `element-type` attribute is used to specify a more restrictive type than the field definition in the class. This might be required in order to map the field to the datastore. It follows the same rules as `element-type` in element `collection`.

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 persistence-capable types, the `Object` type, interface types and multi-valued types; and `"true"` for primitive types, wrapper types, and single-valued `Object` types including `String`, `Date`, and `Locale`. It also defaults to `true` if the `element` element contains the `embedded` element specifying the embedded mapping of the element.

The `dependent-element` attribute indicates that the array element contains a reference that is to be deleted if the referring instance is deleted, the array is replaced, or the reference is nullified.

The `serialized-element` attribute indicates that the array element is to be serialized for storage using `writeObject`, and cannot be queried.

The attributes `serialized-element=”true”` and `embedded-element=”true”`are mutually exclusive.

The embedded treatment of the array instance itself is governed by the `embedded` attribute of the `field` element.

18.15.4 ELEMENT embedded

The `embedded` element specifies the mapping for an embedded type. It contains multiple field and property elements, one for each field and property in the type.

The `null-indicator-column` attribute optionally identifies the name of the column used to indicate whether the embedded instance is null. By default, if the value of this column is null, then the embedded instance is null. This column might be mapped to a field or property of the embedded instance but might be a synthetic column for the sole purpose of indicating a null reference.

The `null-indicator-value` attribute specifies the value to indicate that the embedded instance is null. This is only used for non-nullable columns.

If `null-indicator-column` is omitted, then the embedded instance is assumed always to exist.

The `owner-field` attribute specifies the name of the field or property in the embedded type that should contain a reference to the owning instance. This field or property is automatically instantiated by the implementation, and is not mapped to anything in the data store.

18.15.5 ELEMENT key

This element specifies the mapping for the key component of a `Map` field.

If only one column is mapped, and no additional information is needed for the column, then the `column` attribute can be used. Otherwise, the `column` element(s) are used.

If the `Map` field is mapped using the `mapped-by` attribute in the field metadata, then the key can be mapped to a field in the same class. In this case, use the `mapped-by` attribute in the key metadata to name the field containing the key data.

The `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.

The `delete-action`, `update-action`, `indexed`, `and unique` attributes specify the characteristics of a constraint to be generated.

18.15.6 ELEMENT value

This element specifies the mapping for the value component of a `Map` field.

If only one column is mapped, and no additional information is needed for the column, then the column attribute can be used. Otherwise, the `column` element(s) are used.

If the `Map` field is mapped using the `mapped-by` attribute in the field metadata, then the value can be mapped to a field in the same class. In this case, use the `mapped-by` attribute in the value metadata to name the field containing the value data.

The `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.

The `delete-action`, `update-action`, `indexed`, `and unique` attributes specify the characteristics of a constraint to be generated.

18.15.7 ELEMENT element

This element specifies the mapping for the element component of arrays and collections.

If only one column is mapped, and no additional information is needed for the column, then the column attribute can be used. Otherwise, the `column` element(s) are used.

The `mapped-by` attribute specifies that the field is mapped to the same database column(s) as the named field in the other class.

The `delete-action`, `update-action`, `indexed`, `and unique` attributes specify the characteristics of a constraint to be generated.

18.15.8 ELEMENT order

This element specifies the mapping for the ordering component of arrays and lists.

If no additional information is needed for the ordering column except for the name, then the column attribute can be used. Otherwise, the `column` element(s) are used.

If the array or list field is mapped using the `mapped-by` attribute in the field metadata, then the ordering can be mapped to a field in the same class. In this case, use the `mapped-by` attribute in the order metadata to name the field containing the ordering data.

The `serialized` attribute specifies that the key values are to be serialized into the named column.

18.16 ELEMENT query

This element specifies the serializable components of a query. Queries defined using metadata are used with the `newNamedQuery` method of `PersistenceManager`.

The `name` attribute specifies the name of the query, and is required.

The `language` attribute specifies the language of the query. The default is `“javax.jdo.query.JDOQL”.` To specify SQL, use `“javax.jdo.query.SQL”`. Names for languages other than these are not standard.

The `unmodifiable` attribute specifies whether the query can be modified by the program.

The body of the `query` element specifies the text of the query. This is the single-string query as defined in section 14.6.13. For convenience, single quotes can be used to delimit string constants in the filter.

For SQL queries, in which it is not possible to specify uniqueness and the result class in the query itself, the attributes `unique` and `result-class` can be used.

18.17 ELEMENT sequence

The `sequence` element identifies a sequence number generator that can be used for several purposes:

- by the JDO implementation to generate application identity primary key values;
- by the JDO implementation to generate datastore identity primary key values;
- by the JDO implementation to generate non-key field values;
- by an application to generate unique identifiers for application use.

The `name` attribute specifies the name for the sequence number generator.

The `strategy` attribute specifies the strategy for generating sequence numbers. Standard values are:

- `nontransactional`: values are obtained outside of the transaction
- `transactional`: values are obtained in a transaction; if the transaction rolls back, gaps might occur in the sequence numbers
- `contiguous`: values are obtained in a transaction; all sequence numbers are guaranteed to be used. This implies that use of the sequence is serialized by transactions.

The `datastore-sequence` attribute names the sequence used to generate key values. This must correspond to a named sequence in the database schema.

The `factory-class` attribute names the user-defined class of the factory for the sequence. The class must have a static method `newInstance()` that returns an instance of `Sequence`. This method will be invoked once per named sequence per `PersistenceManagerFactory` and the same instance will be used for every reference to the same named sequence in the context of that `PersistenceManagerFactory`.

This element is used in conjunction with the `getSequence(String name)` method in `PersistenceManager`. The `name` parameter is the fully qualified name of the sequence.

18.18 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.

18.19 ELEMENT orm

This element specifies mapping information. It is the top-level element in a mapping file whose extension is .orm. Many of the same elements in the jdo document are valid for orm.

18.20 ELEMENT jdoquery

This element specifies named query information separate from the persitence and mapping metadata. It is the top-level element in a file whose extension is .jdoquery. Many of the same elements in the jdo document are valid for jdoquery.

18.21 The jdo Document Type Descriptor

This describes files stored as .jdo files.

Note: The document type descriptors are descriptive, not normative. The xml schema in the binary distribution is normative.

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 2.0//EN"`; a jdo implementation might optimize access for this id.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The DOCTYPE should be as follows for metadata documents.
<!DOCTYPE jdo
    PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Meta-
data 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
-->
<!ELEMENT jdo (extension*, (package|query)+, extension*)>
<!ATTLIST jdo catalog CDATA #IMPLIED>
<!ATTLIST jdo schema CDATA #IMPLIED>

<!ELEMENT package (extension*, (interface|class|sequence)+, exten-
sion*)>
<!ATTLIST package name CDATA ''>
<!ATTLIST package catalog CDATA #IMPLIED>
<!ATTLIST package schema CDATA #IMPLIED>

<!ELEMENT interface (extension*, datastore-identity?, primary-
key?, inheritance?, version?, join*, foreign-key*, index*, unique*,
property*, query*, fetch-group*, extension*)>
<!ATTLIST interface name CDATA #REQUIRED>
<!ATTLIST interface table CDATA #IMPLIED>
<!ATTLIST interface identity-type (datastore|application|nondura-
ble) #IMPLIED>
<!ATTLIST interface objectid-class CDATA #IMPLIED>
<!ATTLIST interface requires-extent (true|false) 'true'>
<!ATTLIST interface detachable (true|false) 'false'>
<!ATTLIST interface embedded-only (true|false) #IMPLIED>
<!ATTLIST interface catalog CDATA #IMPLIED>
<!ATTLIST interface schema CDATA #IMPLIED>

<!ELEMENT property (extension*, (array|collection|map)?, join?,
embedded?, element?, key?, value?, order?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property persistence-modifier (persistent|transaction-
al|none) #IMPLIED>
<!ATTLIST property default-fetch-group (true|false) #IMPLIED>
<!ATTLIST property load-fetch-group CDATA #IMPLIED>
<!ATTLIST property null-value (default|exception|none) 'none'>
<!ATTLIST property dependent (true|false) #IMPLIED>
<!ATTLIST property embedded (true|false) #IMPLIED>
<!ATTLIST property primary-key (true|false) 'false'>
<!ATTLIST property value-strategy CDATA #IMPLIED>
<!ATTLIST property sequence CDATA #IMPLIED>
```

```
<!ATTLIST property serialized (true|false) #IMPLIED>
<!ATTLIST property table CDATA #IMPLIED>
<!ATTLIST property column CDATA #IMPLIED>
<!ATTLIST property delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST property indexed (true|false|unique) #IMPLIED>
<!ATTLIST property unique (true|false) #IMPLIED>
<!ATTLIST property mapped-by CDATA #IMPLIED>
<!ATTLIST property recursion-depth CDATA #IMPLIED>
<!ATTLIST property field-name CDATA #IMPLIED>

<!ELEMENT class (extension*, implements*, datastore-identity?, primary-key?, inheritance?, version?, join*, foreign-key*, index*, unique*, column*, field*, property*, query*, fetch-group*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|nondurable) #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class table CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>
<!ATTLIST class detachable (true|false) 'false'>
<!ATTLIST class embedded-only (true|false) #IMPLIED>
<!ATTLIST class persistence-modifier (persistence-capable|persistence-aware|non-persistent) #IMPLIED>
<!ATTLIST class catalog CDATA #IMPLIED>
<!ATTLIST class schema CDATA #IMPLIED>

<!ELEMENT primary-key (extension*, column*, extension*)>
<!ATTLIST primary-key name CDATA #IMPLIED>
<!ATTLIST primary-key column CDATA #IMPLIED>

<!ELEMENT join (extension*, primary-key?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join outer (true|false) 'false'>
<!ATTLIST join delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>

<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #IMPLIED>
<!ATTLIST version column CDATA #IMPLIED>
<!ATTLIST version indexed (true|false|unique) #IMPLIED>
```

```
<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ATTLIST datastore-identity strategy CDATA 'native'>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>

<!ELEMENT implements (extension*, property*, extension*)>
<!ATTLIST implements name CDATA #REQUIRED>

<!ELEMENT inheritance (extension*, join?, discriminator?, extension*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>

<!ELEMENT discriminator (extension*, column*, index?, extension*)>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>

<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>
<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null (true|false) #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ATTLIST column insert-value CDATA #IMPLIED>

<!ELEMENT field (extension*, (array|collection|map)?, join?, embedded?, element?, key?, value?, order?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier (persistent|transactional|none) #IMPLIED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field column CDATA #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>
<!ATTLIST field serialized (true|false) #IMPLIED>
<!ATTLIST field dependent (true|false) #IMPLIED>
<!ATTLIST field value-strategy CDATA #IMPLIED>
<!ATTLIST field delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
```

```
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>
<!ATTLIST field load-fetch-group CDATA #IMPLIED>
<!ATTLIST field recursion-depth CDATA #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>

<!ELEMENT foreign-key (extension*, (column* | field* | property*), extension*)>
<!ATTLIST foreign-key table CDATA #IMPLIED>
<!ATTLIST foreign-key deferred (true|false) #IMPLIED>
<!ATTLIST foreign-key delete-action (restrict|cascade|null|default|none) 'restrict'>
<!ATTLIST foreign-key update-action (restrict|cascade|null|default|none) 'restrict'>
<!ATTLIST foreign-key unique (true|false) #IMPLIED>
<!ATTLIST foreign-key name CDATA #IMPLIED>

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

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

<!ELEMENT key (extension*, embedded?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST key delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST key update-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>
<!ATTLIST key mapped-by CDATA #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST value column CDATA #IMPLIED>
```

```
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value delete-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST value update-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>
<!ATTLIST value mapped-by CDATA #IMPLIED>

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

<!ELEMENT element (extension*, embedded?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST   element   delete-action   (restrict|cascade|null|de-
fault|none) #IMPLIED>
<!ATTLIST   element   update-action   (restrict|cascade|null|de-
fault|none) #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>
<!ATTLIST element mapped-by CDATA #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order mapped-by CDATA #IMPLIED>

<!ELEMENT fetch-group (extension*, (fetch-group|field|property)*,
extension*)>
<!ATTLIST fetch-group name CDATA #REQUIRED>
<!ATTLIST fetch-group post-load (true|false) #IMPLIED>

<!ELEMENT embedded (extension*, (field|property)*, extension*)>
<!ATTLIST embedded owner-field CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-value CDATA #IMPLIED>

<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ATTLIST  sequence  strategy  (nontransactional|contiguous|noncon-
tiguous) #REQUIRED>
```

```
<!ELEMENT index (extension*, (column* | field* | property*), exten-
sion*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) #IMPLIED>
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT unique (extension*, (column* | field* | property*), ex-
tension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

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

18.22 The orm Document Type Descriptor

This describes files stored as .orm files.

Note: The document type descriptors are descriptive, not normative. The xml schema in the binary distribution is normative.

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/orm.dtd`"
- If PUBLIC is used, the public id should be "`-//Sun Microsystems, Inc.// DTD Java Data Objects Mapping Metadata 2.0//EN`"; a jdo implementation might optimize access for this id.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The DOCTYPE should be as follows for metadata documents.
<!DOCTYPE orm
   PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Mapping
Metadata 2.0//EN"
    "http://java.sun.com/dtd/orm_2_0.dtd">
-->
<!ELEMENT orm (extension*, (package|query)+, extension*)>
<!ATTLIST orm catalog CDATA #IMPLIED>
```

```
<!ATTLIST orm schema CDATA #IMPLIED>

<!ELEMENT package (extension*, (interface|class|sequence)+, extension*)>
<!ATTLIST package name CDATA ''>
<!ATTLIST package catalog CDATA #IMPLIED>
<!ATTLIST package schema CDATA #IMPLIED>

<!ELEMENT interface (extension*, datastore-identity?, primary-key?, inheritance?, version?, join*, foreign-key*, index*, unique*, property*, query*, extension*)>
<!ATTLIST interface name CDATA #REQUIRED>
<!ATTLIST interface table CDATA #IMPLIED>
<!ATTLIST interface catalog CDATA #IMPLIED>
<!ATTLIST interface schema CDATA #IMPLIED>

<!ELEMENT property (extension*, join?, embedded?, element?, key?, value?, order?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property value-strategy CDATA #IMPLIED>
<!ATTLIST property sequence CDATA #IMPLIED>
<!ATTLIST property serialized (true|false) #IMPLIED>
<!ATTLIST property table CDATA #IMPLIED>
<!ATTLIST property column CDATA #IMPLIED>
<!ATTLIST property delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST property indexed (true|false|unique) #IMPLIED>
<!ATTLIST property unique (true|false) #IMPLIED>
<!ATTLIST property mapped-by CDATA #IMPLIED>

<!ELEMENT class (extension*, datastore-identity?, primary-key?, inheritance?, version?, join*, foreign-key*, index*, unique*, column*, field*, property*, query*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class table CDATA #IMPLIED>
<!ATTLIST class catalog CDATA #IMPLIED>
<!ATTLIST class schema CDATA #IMPLIED>

<!ELEMENT primary-key (extension*, column*, extension*)>
<!ATTLIST primary-key name CDATA #IMPLIED>
<!ATTLIST primary-key column CDATA #IMPLIED>

<!ELEMENT join (extension*, primary-key?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join outer (true|false) 'false'>
```

```
<!ATTLIST join delete-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>

<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #IMPLIED>
<!ATTLIST version column CDATA #IMPLIED>
<!ATTLIST version indexed (true|false|unique) #IMPLIED>

<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ATTLIST datastore-identity strategy CDATA 'native'>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>

<!ELEMENT implements (extension*, property*, extension*)>
<!ATTLIST implements name CDATA #REQUIRED>

<!ELEMENT inheritance (extension*, join?, discriminator?, exten-
sion*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>

<!ELEMENT discriminator (extension*, column*, index?, extension*)>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>

<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>
<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null (true|false) #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ATTLIST column insert-value CDATA #IMPLIED>

<!ELEMENT field (extension*, join?, embedded?, element?, key?, val-
ue?, order?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field column CDATA #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field value-strategy CDATA #IMPLIED>
```

```
<!ATTLIST field delete-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>

<!ELEMENT foreign-key (extension*, (column* | field* | property*),
extension*)>
<!ATTLIST foreign-key table CDATA #IMPLIED>
<!ATTLIST foreign-key deferred (true|false) #IMPLIED>
<!ATTLIST  foreign-key  delete-action  (restrict|cascade|null|de-
fault|none) 'restrict'>
<!ATTLIST  foreign-key  update-action  (restrict|cascade|null|de-
fault|none) 'restrict'>
<!ATTLIST foreign-key unique (true|false) #IMPLIED>
<!ATTLIST foreign-key name CDATA #IMPLIED>

<!ELEMENT key (extension*, embedded?, column*, foreign-key?, in-
dex?, unique?, extension*)>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST  key  delete-action  (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST  key  update-action  (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>
<!ATTLIST key mapped-by CDATA #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?, in-
dex?, unique?, extension*)>
<!ATTLIST value column CDATA #IMPLIED>
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value delete-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST value update-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>
<!ATTLIST value mapped-by CDATA #IMPLIED>

<!ELEMENT element (extension*, embedded?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST   element   delete-action   (restrict|cascade|null|de-
fault|none) #IMPLIED>
```

```
<!ATTLIST element update-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>
<!ATTLIST element mapped-by CDATA #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order mapped-by CDATA #IMPLIED>

<!ELEMENT embedded (extension*, (field|property)*, extension*)>
<!ATTLIST embedded owner-field CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-value CDATA #IMPLIED>

<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ATTLIST sequence strategy (nontransactional|contiguous|noncontiguous) #REQUIRED>

<!ELEMENT index (extension*, (column* | field* | property*), extension*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) #IMPLIED>
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT unique (extension*, (column* | field* | property*), extension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

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

18.23 The jdoquery Document Type Descriptor

This describes files stored as .jdoquery files.

Note: The document type descriptors are descriptive, not normative. The xml schema in the binary distribution is normative.

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/jdoquery.dtd"`
- If PUBLIC is used, the public id should be `"-//Sun Microsystems, Inc.//DTD Java Data Objects Query Metadata 2.0//EN"`; a jdo implementation might optimize access for this id.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The DOCTYPE should be as follows for jdoquery documents.
<!DOCTYPE jdoquery
    PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Query
Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdoquery_2_0.dtd">
-->
<!ELEMENT jdoquery (extension*, (package|query)+, (extension)*)>

<!ELEMENT package (extension*, (interface|class)+, (extension)*)>
<!ATTLIST package name CDATA ''>

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

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

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) #IMPLIED>
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
```

```
<!ATTLIST extension value CDATA #IMPLIED>
```

18.24 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 a 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">
<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 Extent

This chapter specifies the `Extent` contract between an application component and the JDO implementation.

19.1 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.

19.2 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.

19.3 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 includes subclasses.

```
Class getCandidateClass();
```

This method returns the class of the instances contained in it.

```
FetchPlan getFetchPlan();
```

This method returns the fetch plan associated with the `Extent`. The fetch plan originally is a copy of the fetch plan that is active at the time the `Extent` is obtained from the `PersistenceManager`, and thereafter can be changed independent of the fetch plan of the `PersistenceManager`.

The fetch plan settings affect iterators obtained from the `Extent`. When instances are retrieved from the datastore, the fetch plan settings that are current in the `Extent` affect the retrieval. Note that this means that portable applications do not change the fetch plan of an `Extent` while an iterator is active.

```
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()`.

Any change made to the fetch plan of the associated `PersistenceManager` affects instance retrievals via `next()`. Only instances not already in memory are affected by the `PersistenceManager`'s fetch plan. Fetch plan is described in Section 12.7.

20 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.

20.1 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.

20.1.1 Optimistic Transactions

Optimistic transactions are enabled by the `PersistenceManagerFactory` or `Transaction` method `setOptimistic(true)`. JDO implementations that do not support optimistic transactions throw `JDOUnsupportedOptionException`.

20.1.2 Nontransactional Read

Nontransactional read is enabled by the `PersistenceManagerFactory` or `Transaction` method `setNontransactionalRead(true)`. JDO implementations that do not support nontransactional read throw `JDOUnsupportedOptionException`.

20.1.3 Nontransactional Write

Nontransactional write is enabled by the `PersistenceManagerFactory` or `Transaction` method `setNontransactionalWrite(true)`. JDO implementations that do not support nontransactional write throw `JDOUnsupportedOptionException`.

20.1.4 Transient Transactional

Transient transactional instances are created by the `PersistenceManager` `makeTransactional(Object)`. JDO implementations that do not support transient transactional throw `JDOUnsupportedOptionException`.

20.1.5 RetainValues

A portable application should run the same regardless of the setting of the `retainValues` flag.

20.1.6 IgnoreCache

A portable application should set this flag to `false`. The results of iterating `Extent`s and executing queries might be different among different implementations.

20.2 Object Model

References among persistence-capable 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".

20.3 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.

20.4 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(Map)` method.

20.5 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 persistence-capable 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 only use persistent or public final static field names in filter expressions.

Portable queries must pass persistent or transactional instances as parameters of persistence-capable types.

Wild card queries must use “matches” with a regular expression including only “(?i)” for case-insensitivity, “.” for matching a single characters, and “.*” for matching multiple characters.

20.6 XML metadata

Portable applications will define all persistence-capable classes in the XML metadata.

20.7 Life cycle

Portable applications will not depend on requiring instances to be hollow or persistent-nontransactional, or to remain non-transactional in a transaction.

20.8 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.

20.9 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).

20.10 Binary Compatibility

Portable applications must not use the PersistenceCapable interface. Compliant implementations might use persistence-capable classes that do not implement the PersistenceCapable interface. Instances can be queried as to their state by using the methods in JDOHelper.

Readers primarily interested in developing applications with the JDO API can ignore the following chapters. Skip to 23 – JDOPermission.

21 JDO Reference Enhancer

This chapter specifies the JDO Reference Enhancement, which specifies the contract between JDO persistence-capable classes and JDO StateManager in the binary-compatible 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.

Enhancement is just one possible strategy for JDO implementations. If a JDO implementation supports BinaryCompatibility, it must support the `PersistenceCapable` contract. Otherwise, it need only support the rest of the user-visible contracts (e.g. PersistenceManagerFactory, PersistenceManager, Query, Transaction, and Extent).

> *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.*

21.1 Overview

The JDO Reference Enhancer will be used to modify each persistence-capable class before using that persistence-capable class with the Reference Implementation `PersistenceManager` 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.

21.2 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

21.3 Enhancement: Architecture

The reference enhancement of persistence-capable 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 persistence-capable classes, all methods and fields added to the persistence-capable 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 persistence-capable 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.

Enhancement also treats `Detachable` classes specially, by adding fields and methods needed to manage the detached state of the instance.

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 `void jdoCopyField(Object other, int fieldNumber)` 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 the least-derived class marked as Detachable:

- adds “`implements javax.jdo.spi.Detachable`” to the class definition;
- adds a serializable field “`Object[] jdoDetachedState`” to the class definition;
- adds a method “`void jdoReplaceDetachedState()` to the class definition.

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 two methods, `makeDirty(String fieldName)` and `makeDirty(int fieldNumber)`, to manage making fields dirty.
- 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.

21.4 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.

21.5 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`.

21.6 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.

Classes marked as `Detachable` are not serialization-compatible with un-enhanced classes. This is intentional, and requires that the enhanced version of the class be used wherever the instance might be instantiated. If a `Detachable` class were used in an environment where the un-enhanced class was not used, then access to unloaded fields would not be restricted, and field modifications could not be tracked.

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.]

For `Detachable` classes, the `jdoPreSerialize` method must also initialize the `jdoDetachedState` instance so that the detached state is serialized along with the instance.

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 datastore.

For non-`Detachable` classes, 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.

For `Detachable` classes, the results of restoring a serialized persistent instance graph is a graph of interconnected detached instances that might be attached via the `attachCopy` methods.

21.7 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.

The clone method is also modified to copy the `jdoDetachedState` field of an instance of a `Detachable` class to the clone if the instance is detached. The effect of this is that while detached, the clone is also a detached object. While attached, the detached state will not be cloned, and the clone will therefore be transient.

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.

21.8 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.

21.9 Field Modifiers

Fields in persistence-capable 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 persistence-capable 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.

21.9.1 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.

21.9.2 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).

21.9.3 Persistent

Persistent fields are fields whose values are synchronized with values in the datastore. The synchronization is performed transparent to the methods in the persistence-capable 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.

21.9.4 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.

21.9.5 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 persistence-capable field types to embedded objects (aggregation by containment pattern).

21.9.6 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”.

21.10 Treatment of standard Java field modifiers

21.10.1 Static

Static fields are ignored by the enhancer. They are not initialized by JDO; accesses to values are not mediated.

21.10.2 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.

21.10.3 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.

21.10.4 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.

21.11 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.

21.12 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 these 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.

Regardless of the `jdoFlags` setting, detached instances will disallow access to non-key fields that are not marked as loaded in the detached state.

21.13 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.

21.14 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 field.

Table 9: 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

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
```

21.15 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;
```

This field contains the detached state, if this instance is detached.

```
protected Object[] jdoDetachedState;
```

21.16 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 superclasses 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.

21.17 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 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 boolean jdoIsDetached();`

This method checks if the instance is detached. If so, it returns `true`.

`public final boolean jdoIsDirty();`

This method checks if the instance is detached. If so, it returns the modified state of the field from the detached state. If not, it checks if the `jdoStateManager` field is `null`. If so, it returns `false`. If not, it delegates to the corresponding method in `StateManager`.

`public final void jdoMakeDirty (String fieldName);`

`public final void jdoMakeDirty (int fieldNumber);`

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 jdoGetVersion();`

`public final Object jdoGetTransactionalObjectId();`

These methods check if the instance is detached. If so, they return the appropriate element of the detached state. If not detached, the 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 (StateManager 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.

21.18 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 persistence-capable 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 persistence-capable 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 persistence-capable 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 persistence-capable 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(Object obj);`

In the case of single field identity, the parameter is an instance of one of the following:

- `Number`: the parameter is converted to the appropriate type and passed to the constructor of the single-field identity class
- `String`: the parameter is converted to the appropriate type and passed to the constructor of the single-field identity class
- `ObjectIdFieldSupplier`: the parameter is used to fetch the value of the object id field which is passed to the constructor of the single-field identity class
- `Object`: for `ObjectIdentity` only, the parameter is passed to the constructor of `ObjectIdentity`

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.

21.19 Generated method in least-derived `Detachable` classes

`public void jdoReplaceDetachedState();`

This method delegates to the `jdoStateManager` method `replacingDetachedState`, passing the current value of the detached state and replacing the detached state with the result of the method call.

21.20 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 and whether the class is marked as `Detachable`:

- If the instance is detached, the method checks to see if the field is marked as loaded in the detached state. If the field is not loaded ,then `JDODetachedFieldAccessException` is thrown.
- If the field is `CHECK_READ`, then the method first checks to see if `jdoFlags` field is anything except `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 and whether the class is marked as `Detachable`:

- If the instance is detached, the method checks to see if the field is marked as loaded in the detached state. If the field is not loaded ,then `JDODetachedFieldAccessException` is thrown. If the field is loaded, then the field is marked as modified in the detached state.
- 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 jdoReplaceFields (int[] fields);`This method internally calls `jdoReplaceField` for each field number in the parameter.

`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 jdoProvideFields (int[] fields);`This method internally calls `jdoProvideField` for each field number in the parameter.

`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.

`public final void jdoCopyField (<class> other, int fieldNumber);`

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.

21.21 Example class: Employee

The following class definitions for persistence capable classes are used in the examples. The `Employee` class is enhanced for application identity using `IntIdentity` as the object id class.

```
package com.xyz.hr;
import javax.jdo.spi.*; // generated by enhancer...
class Employee
   implements Detachable // 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
```

21.21.1 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;
*/
```

21.21.2 Generated static initializer

```
static {
javax.jdo.spi.JDOImplHelper.registerClass (
    Employee.class,
    jdoFieldNames,
    jdoFieldTypes,
    jdoFieldFlags,
    jdoPersistenceCapableSuperclass,
    new Employee());
}
```

21.21.3 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 boolean jdoIsDetached(){
   return jdoStateManager==null?false:
      jdoStateManager.isDetached(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);
}
```

21.21.4 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;
}
}
```

21.21.5 Generated jdoReplaceFlags

```
public final void jdoReplaceFlags () {
   if (jdoStateManager != null) {
      jdoFlags = jdoStateManager.replacingFlags (this);
   }
}
```

21.21.6 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;
}
```

21.21.7 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.

The implementation for topmost classes in the hierarchy:

```
protected static int jdoGetManagedFieldCount () {
   return jdoFieldNames.length;
}
```

The implementation for subclasses:

```
protected static int jdoGetManagedFieldCount () {
   return <pc-superclass>.jdoGetManagedFieldCount() +
        jdoFieldNames.length;
}
```

21.21.8 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;
   }
}
```

21.21.9 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;
   }
}
```

21.21.10 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);
   }
}
```

21.21.11 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);
   }
}
```

21.21.12 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 rollback, 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
```

21.21.13 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 ();
}
```

21.21.14 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);
}
```

21.21.15 Generated jdoNewObjectIdInstance

The generated methods create and return a new instance of the object id class.

```
public Object jdoNewObjectIdInstance() {
   return new IntIdentity(Employee.class, empid);
}
public Object jdoNewObjectIdInstance(Object obj) {
   if (obj instanceof String) {
      return new IntIdentity(Employee.class, (String)str);
   } else if (obj instanceof Integer) {
      return new IntIdentity(Employee.class, (Integer)obj);
   } else if (obj instanceof ObjectIdFieldSupplier) {
      return new IntIdentity(Employee.class,
         ((ObjectIdFieldSupplier)obj).fetchIntField(2));
   } else
      throw new JDOUserException(“illegal object id type”);
}
```

21.21.16 Generated jdoCopyKeyFieldsToObjectId

The generated methods copy key field values from the `PersistenceCapable` instance or from the `ObjectIdFieldSupplier`.

```
public void jdoCopyKeyFieldsToObjectId (ObjectIdFieldSupplier fs,
Object oid) {
   throw new JDOFatalInternalException(“Object id is immutable”);
}
public void jdoCopyKeyFieldsToObjectId (Object oid) {
   throw new JDOFatalInternalException(“Object id is immutable”);
}
```

21.21.17 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, ((IntIdentity)oid).getKey());
}
```

This method is part of the PersistenceCapable contract. It copies key fields from the object id instance to the `ObjectIdFieldConsumer`.

```
protected void jdoCopyKeyFieldsFromObjectId (Object oid) {
   empid = ((IntIdentity)oid).getKey());
}
```

This method is used internally to copy key fields from the object id instance to a newly created `PersistenceCapable` instance.

21.21.18 Generated `Detachable` methods

```
public void jdoReplaceDetachedState() {
   jdoDetachedState = sm.replacingDetachedState(this,
      jdoDetachedState);
}
} // end class definition
```

22 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.*

22.1 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.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.jdoReplaceStateManager(null); return true`.

```
package javax.jdo.spi;
public interface StateManager {
```

22.2 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);
```

When the current `StateManager` is not `null`, it should be the only caller of `PersistenceCapable.jdoReplaceStateManager`, which calls this method. This method should be called 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
1: LOAD_REQUIRED
2: DETACHED
```

```
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.

22.3 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);
```

22.4 Dirty management

The following methods provide for marking the `PersistenceCapable` instance dirty:

```
public void makeDirty (PersistenceCapable pc, String field-
Name);public void makeDirty (PersistenceCapable pc, int fieldNum-
ber);
```

22.5 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);
```

22.6 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.

22.7 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.

22.8 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 `isLoaded`.
- 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 `isLoaded` 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 `isLoaded` 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 isLoaded (PersistenceCapable pc, int field);
```

22.8.1 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 currentValue);

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, Object currentValue);
```

22.8.2 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 currentValue, 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, double currentValue, double newValue);
public void setStringField (PersistenceCapable pc, int field, String currentValue, String newValue);
public void setObjectField (PersistenceCapable pc, int field, Object currentValue, Object newValue);
```

22.8.3 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);
```

22.8.4 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);
```

22.9 Detached instance support

```
public Object[] replacingDetachedState (
   Detached pc, Object[] loaded);
```

This method is called by a detachable instance in response to the `StateManager` calling `replaceDetachedState`. It provides the `Detachable` and the detached state to the `StateManager`.

23 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 four 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.jdoReplaceStateManager` 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`.`manageMetadata`: this permission allows a caller to unload metadata for a class or a class loader, thereby releasing resources. This permission is checked by the `unregisterClass()` and `unregisterClasses()` methods of `JDOImplHelper`.

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";
 permission javax.jdo.spi.JDOPermission "manageMetadata";
};
```

24 JDOQL BNF

Grammar Notation

The grammar notation is taken from the Java Language Specification, section 2.4 Grammar Notation.

- Terminal symbols are shown in bold fixed width font 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 subscripted 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.

Single-String JDOQL

This section describes the syntax of single-string JDOQL.

SingleStringJDOQL:
 Select *From*$_{\text{opt}}$ *Where*$_{\text{opt}}$ *Decls* *Grouping*$_{\text{opt}}$ *Ordering*$_{\text{opt}}$ *Range*$_{\text{opt}}$

Select:
 `select` **`unique`**$_{\text{opt}}$ *ResultClause*$_{\text{opt}}$ *IntoClause*$_{\text{opt}}$

IntoClause:
 `into` *ResultClassName*

From:
 `from` *CandidateClassName* *ExcludeClause*$_{\text{opt}}$

ExcludeClause:
 `exclude subclasses`

Where:
 `where` *Expression*

Decls:
 Variables$_{\text{opt}}$ *Parameters*$_{\text{opt}}$ *Imports*$_{\text{opt}}$

Variables:
 `variables` *VariableList*

Parameters:
parameters *ParameterList*

Imports:
ImportList

Grouping:
group by *GroupingClause*

Ordering:
order by *OrderingClause*

Range:
range *Expression* , *Expression*

Filter Specification

This section describes the syntax of the setFilter argument.

Basically, the query filter expression is a Java boolean expression, where some of the Java operators are not permitted. Specifically, pre- and post- increment and decrement (++ and - -), shift (>> and <<) and assignment expressions (+=, -=, etc.) are not permitted.

The Nonterminal *InfixOp* lists the valid operators for binary expressions in decreasing precedence. Operators one the same line have the same precedence. As in Java operators require operands of appropriate types. See the Java Language Specification for more information.

Plase note, the grammar allows arbitrary method calls (see *MethodInvocation*), where JDO only permits the following methods:

Collection methods	`contains(Object),isEmpty(), size()`
Map methods	`containsKey(Object),containsValue(Object), isEmpty(), size(), get()`

String methods	`startsWith(String),endsWith(String),`
	`matches(String),`
	`toLowerCase(),toUpperCase(),`
	`indexOf(String),indexOf(String, int),`
	`substring(int),substring(int, int)`

Math methods	`Math.abs(numeric),Math.sqrt(numeric)`
JDOHelper methods	`getObjectId(Object)`

Expression:
UnaryExpression
Expression InfixOp UnaryExpression

InfixOp: *one of*
`* / %`
`+ -`
`> >= < <= instanceof`
`== !=`
`&`
`|`

&&
||

UnaryExpression:
PrefixOp UnaryExpression
(*Type* **)** *UnaryExpression*
Primary

PrefixOp: *one of*
+ - ~ !

Primary:
Literal
VariableName
ParameterName
this
FieldAccess
MethodInvocation
ClassOrInterfaceName
(*Expression* **)**
AggregateExpression [1]

FieldAccess:
FieldName
Primary . *FieldName*

MethodInvocation:
Primary . *MethodName* **(** *ArgumentList*$_{opt}$ **)**

ArgumentList:
Expression
ArgumentList , *Expression*

AggregateExpression:
count (distinct$_{opt}$ *CountArgument* **)**
sum (distinct$_{opt}$ *Expression* **)**
min (*Expression* **)**
max (*Expression* **)**
avg (distinct$_{opt}$ *Expression* **)**

CountArgument:
this
FieldAccess
VariableName

[1] Please note, an *AggregateExpression* is only allowed as part of a result specification or a having specification.

Parameter Declaration

This section describes the syntax of the declareParameters argument.

```
ParameterList:
     Parameters ,opt

ParameterDecls:
     ParameterDecl
     ParameterDecls , ParameterDecl

ParameterDecl:
     Type ParameterName
```

Please note, as a usability feature *ParameterList* supports an optional trailing comma (in addition to what the Java syntax allows in a parameter declaration).

Variable Declaration

This section describes the syntax of the declareVariables argument.

```
VariableList:
     VariableDecls ;opt

VariableDecls:
     VariableDecl
     VariableDecls ; VariableDecl

VariableDecl:
     Type ParameterName
```

Please note, as a usability feature `VariableList` defines the trailing semicolon as optional (in addition to what the Java syntax allows in a variable declaration).

Import Declaration

This section describes the syntax of the declareImports argument.

```
ImportList:
     ImportDecls ;opt

ImportDecls:
     ImportDecl
     ImportDecls ; ImportDecl

ImportDecl:
     import QualifiedIdentifier
     import QualifiedIdentifier . *
```

Please note, as a usability feature `ImportList` defines the trailing semicolon as optional (in addition to what the Java syntax allows in an import statement).

Ordering Specification

This section describes the syntax of the setOrdering argument.

OrderingClause:
OrderingSpecs ,$_{\text{opt}}$

OrderingSpecs:
OrderingSpec
OrderingSpecs , *OrderingSpec*

OrderingSpec:
Expression Ascending
Expression Descending

Ascending: one of
asc ascending

Descending: one of
desc descending

Please note, as a usability feature *OrderingClause* supports an optional trailing comma.

Result Specification

This section describes the syntax of the setResult argument.

ResultClause:
distinct$_{\text{opt}}$ *ResultSpecs* ,$_{\text{opt}}$

ResultSpecs:
ResultSpec
ResultSpecs , *ResultSpec*

ResultSpec:
Expression ResultNaming$_{\text{opt}}$

ResultNaming:
as *Identifier*

Please note, a result specification expression may be an aggregate expression. As a usability feature *ResultClause* supports an optional trailing comma.

Grouping Specification

This section describes the syntax of the setGrouping argument.

GroupingClause:
GroupingSpecs ,$_{\text{opt}}$ *HavingSpec*$_{\text{opt}}$

GroupingSpecs:
Expression
GroupingSpecs , *Expression*

HavingSpec:
having *Expression*

Please note, a having specification expression may include an aggregate expression. As a usability feature *GroupingClause* supports an optional trailing comma.

Types

This section describes a type specification, used in a parameter or variable declaration or in a cast expression.

```
Type
     PrimitiveType
     ClassOrInterfaceName

PrimitiveType:
     NumericType
     boolean

NumericType:
     IntegralType
     FloatingPointType

IntegralType: one of
     byte short int long char

FloatingPointType: one of
     float double
```

Literals

A literal is the source code representation of a value of a primitive type, or the String type. Please refer to the Java Language Specification for the lexical structure of Integer-, Floating Point-, and String-Literals. JDOQL allows String-Literals being enclosed in either single quotes or double quotes. A single character enclosed in either single or double quotes is considered to a be both: a char and a string literal.

```
Literal:
     IntegerLiteral
     FloatingPointLiteral
     BooleanLiteral
     StringLiteral
     NullLiteral

IntegerLiteral: ...

FloatingPointLiteral: ...

BooleanLiteral: one of
     true false

StringLiteral: ...

NullLiteral:
     null
```

Names

A name is a possibly qualified identifier. Please refer to the Java Language Specification for the lexical structure of identifiers.

```
QualifiedIdentifier:
     Identifier
     QualifiedIdentifier . Identifier

CandidateClassName:
     QualifiedIdentifier

ResultClassName:
     QualifiedIdentifier

ClassOrInterfaceName:
     QualifiedIdentifier

VariableName:
     Identifier

ParameterName:
     Identifier
     ColonPrefixedIdentifier

FieldName:
     Identifier

MethodName:
     Identifier
```

Keywords

Keywords must not be used as package names, class names, parameter names, or variable names in queries. Keywords are permitted as field names only if they are on the right side of the `.` in field access expressions as defined in the Java Language Specification second edition, section 15.11. Keywords include the Java language keywords and the JDOQL keywords. Java keywords are as defined in the Java language specification section 3.9, plus the boolean literals true and false, and the null literal. JDOQL keywords maybe written in all lower case or all upper case.

```
JDOQLKeyword: one of
     as          AS          asc         ASC
     ascending   ASCENDING   avg         AVG
     by          BY          count       COUNT
     desc        DESC        descending  DESCENDING
     distinct    DISTINCT    exclude     EXCLUDE
     from        FROM        group       GROUP
     having      HAVING      into        INTO
     max         MAX         min         MIN
     order       ORDER       parameters  PARAMETERS
     range       RANGE       select      SELECT
     subclasses  SUBCLASSES  sum         SUM
     unique      UNIQUE      variables   VARIABLES
     where       WHERE
```

25 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.

25.1 Nested Transactions

Define the semantics of nested transactions.

This proposal is still pending as of JDO 2.0.

25.2 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.

This proposal is still pending as of JDO 2.0.

25.3 Inter-PersistenceManager References

Explain how to establish and maintain relationships between persistent instances managed by different `PersistenceManager`s.

This proposal is still pending as of JDO 2.0.

25.4 Enhancer Invocation API

A standard interface to call the enhancer will be defined.

This proposal is still pending as of JDO 2.0.

25.5 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.

This functionality is now part of JDO 2.0.

25.6 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.

This functionality is now part of JDO 2.0.

25.7 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.

This proposal has been rejected. If this is valuable for persistent instances, it is just as valuable for transient instances. To have the behavior change when making an instance persistent is probably inappropriate.

This proposal should become an independent Java Specification Request.

25.8 Case-Insensitive Query

Use of String.toLowerCase() as a supported method in query filters would allow case-insensitive queries.

This functionality is now part of JDO 2.0.

25.9 String conversion in Query

Supported String constructors String(<integer expression>) and String(<floating-point expression>) would make queries more flexible.

This proposal is still pending as of JDO 2.0.

25.10 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.

25.11 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.

25.12 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.

25.13 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.

This functionality is now part of JDO 2.0.

25.14 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.

25.15 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`.

25.16 Distributed object support

Provide for remote object graph support, including instance reconciliation, relationship graph management, instance insertion ordering, etc.

This functionality is now part of JDO 2.0.

25.17 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. This functionality is now part of JDO 2.0.

26 JDO 1.0.1 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.

26.1 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.

26.2 ELEMENT package

This element includes all classes in a particular package. The complete qualified package name is required.

26.3 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 deprecated for this release. It is ignored so metadata files from previous releases can be used.

26.4 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`: `Currency`, `Date`, `Locale`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedHashMap`, `LinkedHashSet`, `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 persistence-capable 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 persistence-capable 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”`

26.4.1 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 persistence-capable 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.

26.4.2 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 persistence-capable 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.

26.4.3 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 persistence-capable 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.

26.5 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.

26.6 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|transaction-
al|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>
```

26.7 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=”IntIdentity”>
<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>
```

27 Public Feedback Request

This Chapter is devoted to issues for which public feedback is requested. During the Early Draft Review period, the expert group would like the public to provide feedback on these specific issues.

27.1 Annotations for metadata

JSR 14 and 175 are now standard in J2SE 1.5. These language enhancements allow for users to annotate their Java source files with information that in previous releases had to be put into separate metadata files.

The intent for JDO 2.0 is to exploit JSR 14 to obviate the need for metadata defining the types of collection elements and map keys and values. The metadata to define classes as persistent-capable can be embedded in the source file. The combination of these two new features should allow users to avoid the .jdo metadata files completely.

The embedded metadata tags will be included in a future early draft release of the specification.

As of the Public Draft, this proposal has been rejected. Annotations will be specified in JSR 220, and there is not sufficient justification for having a completely different specification for JDO.

27.2 Attach and detach life cycle listener callbacks

Should we add method attach and detach to the life cycle listener interface, allowing the application to monitor attach and detach events?

This proposal has been adopted.

27.3 Proxy support for detached instances

For non-binary-compatible implementations to support the detached instance contract, it must throw a `JDOUserException` if a non-loaded relationship field is accessed while detached.

The JDO package might contain a class suitable as an InvocationHandler for cases where java.lang.reflect.Proxy is used as the strategy. This class would do nothing but throw an exception if it is accessed. This would avoid the requirement that the client have access to vendor-specific classes that implement this behavior.

Support for proxies of references to classes (which cannot be proxied using java.lang.reflect.Proxy) will require additional investigation.

This proposal does not look practical. The strategy only works for reference types that are standard interfaces (e.g. Collection, Set) or classes (HashSet, Hashtable, etc.). Therefore, non-binary-compatible implementations will need to either instantiate single-valued references or include vendor-specific subclasses in the client jar.

27.4 Deleting detached instances

Currently the only way to delete detached instances is to define them as dependent in the metadata of a referencing persistent class. If while detached, the instance “owning” the dependent instance clears the field or removes the dependent instance from a collection, array, or map, then upon reattachment, the dependent instance will be deleted from the datastore.

Allowing a detached instance to be deleted by the application would require changes to the detachment API.

As of the Public Draft, this proposal has been rejected. There are too many issues and not enough justification for adding the complexity to the detachment pattern.

27.5 Implicit variable declarations

JDOQL requires declaring variables in a separate declarations section, in both the API and the metadata. It might be possible to declare them in the filter itself. For example, instead of:

```
query.declareVariables(“Employee e”);
query.setFilter(“emps.contains(e) && e.name ==’George’”);
```

declaring the variable inline:

```
query.setFilter(“emps.contains(Employee e) && e.name ==’George’”);
```

This change might require less user typing, but more JDO implementation analysis to scope the variable.

The concept behind this proposal has been adopted. The variable in the above example can be cast explicitly, as in:

```
query.setFilter
(“emps.contains((Employee) e) && e.name ==’George’”);
```

27.6 Shortcuts for certain JDOQL static methods

Some static methods are defined in JDOQL and currently require the class name and method name to be spelled out. It might be useful to define some shortcuts for these methods:

```
static double Math.sqrt(double): sqrt(double)
static double Math.abs(double): abs(double)
static Object JDOHelper.getObjectId(Object): id(Object)
```

As of the Public Draft, this proposal has been rejected.

27.7 Attribute names for column name

In metadata, a column that has only one attribute, name, could be “promoted” to be an attribute in the containing element. The issue is what to call the attribute. It has been argued that column-name is more descriptive than column for this purpose.

Without column promotion:

```
<field name=”salary”>
    <column name=”SAL”/>
```

</field>

Promotion using “column”:

<field name=”salary” column=”SAL”/>

Promotion using “column-name”:

<field name=”salary” column-name=”SAL”/>

This proposal has been adopted, using the simpler “column” attribute.

27.8 Specification of indexes

Currently indexes are not specified. Where should the definition of indexes be placed? These are needed for many types of datastores, so the definition probably belongs in the JDO metadata (not in mapping metadata).

As of the Public Draft, indexes can be specified in metadata.

27.9 IdGenerator and Sequence are similar

The concepts of IdGenerator and Sequence are very similar. They both are factories for unique primary key values. These two interfaces can be combined; their implementation can be either automatically provided by the JDO vendor, or users can write their own implementation classes.

Similarly, values for non-key fields can be automatically generated by a sequence or other strategy. Both key- and non-key-field values should be definable using a similar notation.

As of the Public Draft, this proposal has been accepted. Generation of ids can be done by specifying a JDO implementation of a sequence or a user-defined sequence.

27.10 Embedded, dependent, and serialized values

There are many strategies for handling mapping of collection, map, and array values. The entire collection, map, or array might be serialized into a column. Alternatively, the keys, values, and elements might be serialized into their own column(s). Or, keys, values, and elements might refer to columns in another table.

Independent of the mapping, the collection, map, and array might be defined as dependent, meaning that if the containing instance removes a reference to it, then it should be removed from the database. And keys, values, and elements might be defined as dependent even if the containing collection, map, or array were not.

In the early draft, these concepts are materialized as attributes of field, collection, map, and array elements. The placement of these attributes and elements need to be rationalized.

As of the Public Draft, this information has been incorporated into the jdo and/or mapping metadata.

27.11 Deprecate dfgOnly parameter?

The `retrieve` methods containing the `dfgOnly` parameter could be deprecated, as there is extensive new capability with fetch groups.

Similarly, detachCopy, refresh, retrieve, and a possible new method makeTransientCopy could have a fetch group explicitly named in the API or they could be defined to use the active fetch groups.

27.12 Fetch Group definition in metadata

Currently, the definition of fields in fetch groups is the same definition as for fields in classes. This may be confusing, and we might rename the field element in fetch-group to be fetched-field or some other name.

The current definition of fetch groups will break some JDO 1.0 applications using refresh() and retrieve(). Refresh and retrieve with fetch groups is arguably better but compatibility is important.

The #key and #value syntax for maps and #element syntax for collections/arrays could be improved. This needs a bit more thought.

27.13 Version information

Currently, the version of an instance is returned as an Object. This might not be the best representation of a version, and it might be better to define an interface, javax.jdo.Version to encapsulate it. This would mean that it would no longer be possible to use a simple type such as Long to represent the version, but it would be type-safe and compile-time checked.

27.14 Single-string JDOQL

As of the Public Draft, the select keyword is required to distinguish single-string JDOQL from a filter. There is still room for discussion here.

27.15 Length, Precision and Scale

As of the Public Draft, the precision of numeric columns and the length of character columns are specified using the same metadata attribute: length. In SQL, the length of numeric columns is specified using the name precision. Should we add precision to the attributes in column to be used instead of length for numeric columns?

27.16 Detachment Contract

As of the Public Draft, detached objects are constructed by explicit API, by serializing persistent graphs, or by closing the PersistenceManager. There are interactions among the serialization contract, fetch groups, and the detachment contract.

We are still examining use cases to see if these can be made consistent.

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

With JDO 2.0, enhancement is now no longer required. Reflection techniques for examining persistent instances at transaction commit can be used instead, and proxies can be used to fault in referenced instances.

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.

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, JDODataStoreException, 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 birthdate. 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 isLoaded.

Added methods PersistenceManager.getTransactionalObjectId, PersistenceCapable.jdoGetTransactionalObjectId, and JDOHelper.getTransactionalObjectId 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 newMapInstance.

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.jdoReplaceStateManager 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 newQuery 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 metadata 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 (ArgumentList$_{opt}$) 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.

C.18 Changes since 1.0.1

In 5.4, added classes used as an application identity class where there is a single application identity field.

In 6.4.3, added interfaces and classes required to be supported as persistent field types: LinkedHashMap, LinkedHashSet, LinkedList, and Currency.

Added to 7.3 .1 a method to retrieve the version of an instance.

Added to 7.4.6 a method to determine if an instance is detached.

Changed 7.12 to add methods handling SimpleIdentity.

Changed in 8.5 the signature of the getPersistenceManagerFactory from Properties to Map.

Added to 8.5 new helper methods for getting PersistenceManagerFactory.

Added to 8.6 new options to specify the mapping for a PersistenceManagerFactory.

Updated 10 to disaggregate instance callbacks.

Changed in 11.1 and 11.7 the parameter of the getPersistenceManagerFactory from Properties to Map.

Changed 11.6 to add javax.jdo.option.BinaryCompatibility, javax.jdo.option.UnconstrainedQueryVariables, javax.jdo.query.SQL, and javax.jdo.option.GetDataStoreConnection to optional features that can be supported by the implementation.

Added to 11.8 a second level cache management API.

Added to 11.9 life cycle event listeners.

Changed requirements for PersistenceCapable to refer to BinaryCompatibility throughout.

Added new method in 12.6.4 getExtent(Class persistenceCapableClass).

Added to 12.6 a discussion on using interfaces with Extents.

Added to 12.6.1 a new method refreshAll(JDOException ex) to refresh instances after a failed optimistic transaction.

Added to 12.6.5 new methods getObjectsById to retrieve multiple instances based on id.

Added to 12.6.5 a new methods getObjectById to retrieve an instance based on class and key.

Added 12.6.6 newInstance method to create instances of persistence-capable interfaces.

Added 12.6.8 methods to detach and attach instances for multi-tier applications.

Added 12.7 methods to specify how instances are fetched from the datastore.

Added 12.8 a method to explicitly flush changes to the datastore.

Added to 12.11 methods to access multiple User Objects.

Added 12.14 new method getSequence.

Added 12.15 new LifecycleEventListener.

Added 12.16 new method getDataStoreConnection.

Clarified 13.4.4 if a transaction is active when begin is called, or a transaction is not active when commit or rollback is called, JDOUserException is thrown.

Added 13.4.5 get/setRollbackOnly to the Transaction interface.

Added to 14.5 newNamedQuery method.

Added to 14.6.1 setParameters methods to bind parameters to query instances.

Added to 14.6.2 the requirement for support of public final static fields in query filters.

Added to 14.6.2 table with supported methods on Collection, Map, and String.

Added to 14.6.2 static method JDOHelper.getObjectId(Object) to allow use of object id in queries.

Added after 14.6.7 new query elements for uniqueness, result, result class, grouping, and result cardinality limits.

Added after 14.6.12 a table for interactions among new query elements.

Added after 14.6 a new section to describe delete by query.

Added after 14.6 a new section to describe support for SQL native queries.

Changed 14.6.6 to permit ordering on boolean fields as a non-portable extension.

Moved Chapter 15 Extent to Chapter 19.

Added new Chapter 15 with object-relational mapping examples.

Moved Chapter 18 to Chapter 25 for JDO 1.0.1 XML metadata.

Added object-relational mapping metadata to Chapter 18.

Added 20.10 to discuss Binary Compatibility portability implications.

Renumbered Chapter 20 Reference Enhancer to Chapter 21.

Added new methods to Chapter 21 to support detached instances.

Updated 21.20.7 to correct a bug in the specification and implementation of getManagedFieldCount.

Renumbered Chapter 21 State Manager to Chapter 22.

Added new methods to Chapter 22 to support detached instances.

Updated 24.6 BLOB/CLOB datatype support to reflect that this functionality is part of JDO 2.0.

Updated 24.8 Case-Insensitive Query to reflect that this functionality is part of JDO 2.0.

Updated 24.13 Projections in query to reflect that this functionality is part of JDO 2.0.

Updated 24.16 Distributed object support to reflect that this functionality is part of JDO 2.0.

Updated 24.17 Object-Relational Mapping to reflect that this functionality is part of JDO 2.0.

Removed B.2 which discussed implications of removing PersistenceCapable.

C.19 Changes since Proposed Final Draft

Updated 18.14 to remove serialized attribute from element, key, and value. Removed foreign-key attribute from field element. Added attribute serialized-element to elements collection and array. Added attributes serialized-key and serialized-value to element map.

Removed attribute serialized from orm metadata. Added serialized-element, serialized-key, and serialized-value to jdo metadata.

Removed true/false use of attribute foreign-key in metadata.

Allowed persistence-capable class to be the parameter of PersistenceManager newInstance.

Allowed attribute order in element collection to permit specifying a column to allow duplicates.

Allowed java.lang classes to be used in metadata without importing them.

Added DetachAllOnCommit property to PersistenceManager to facilitate construction of detached instances.

Changed signature of makePersistent to return the persistent instances, and to attach detached instances.

Added attribute element-type to element array in metadata.

Added field-type to element field in metadata.

Specified behavior of null values in aggregates in JDOQL.

Allowed distinct with aggregates in JDOQL.

Allowed constructors or result class in JDOQL.

Allowed setUnique for delete by query in JDOQL.

Required relationships mapped using mapped-by to be consistent after flush.

Replaced fetch-depth by recursion-depth in fetch plan.

Added methods to specify detachment roots in fetch plan.

Automatically import JDOHelper in JDOQL.

Defined behavior of deletePersistent on detached instances.

Index

Index

Index

Index

Index

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