Java™ Data Objects

JSR 12

Version 1.0.1

Java Data Objects Expert Group

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

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

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

2 Overview

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

2.1 Definitions

2.1.1 JDO common interfaces

JDO Instance

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

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

JDO Implementation

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

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

JDO Enhancer

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

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

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

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

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

Table 1: Which Enhancement Interface is Used

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

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

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

2.1.2 JDO in a managed environment

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

Enterprise Information System (EIS)

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

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

EIS Resource

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

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

Resource Manager (RM)

A resource manager manages a set of shared resources. A client requests access to a resource manager to use its managed resources. A transactional resource manager can 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, update data in an underlying datastore, and do not directly represent data in the datastore.

Entity Beans

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

Helper objects

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

Container

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

2.2 Rationale

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

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

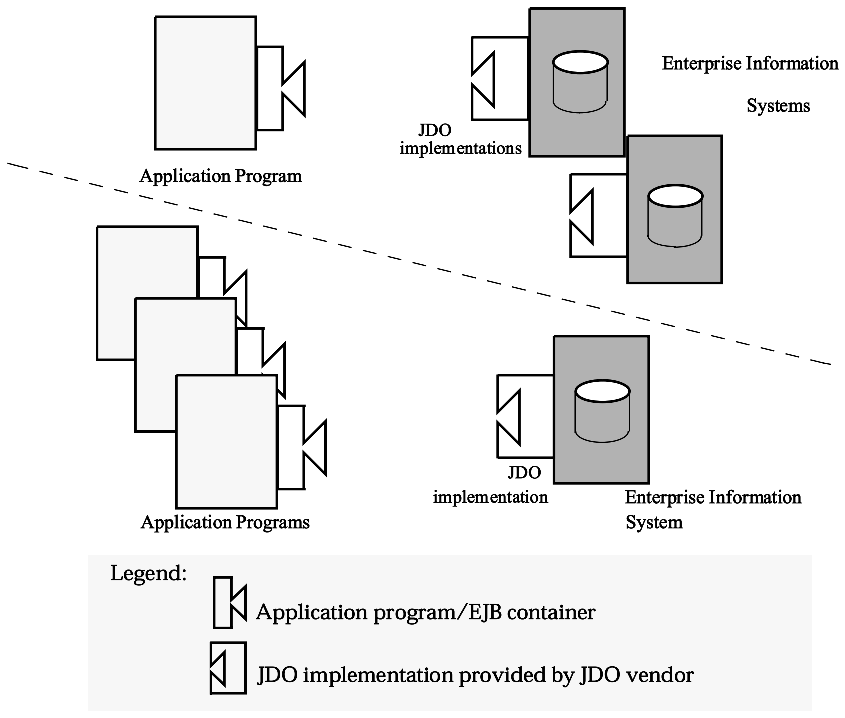
This architecture enables an EIS vendor to provide a standard data access interface for its EIS. The JDO implementation is plugged into an application server and provides underlying infrastructure for integration between the EIS and application components.

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

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

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

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


Enterprise Information Systems
JDO implementations
Application Program
Application Programs
JDO implementation
Enterprise Information System
Legend:
Application program/EJB container
JDO implementation provided by JDO vendor

2.3 Goals

The JDO architecture has been designed with the following goals:

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

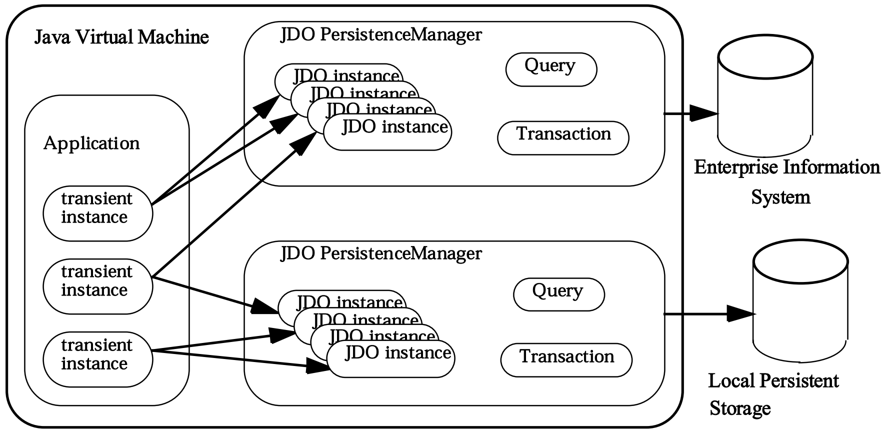
- The JDO architecture simplifies the development of scalable, secure and transactional JDO implementations for a wide range of EISes — ERP systems, database systems, mainframe-based transaction processing systems.
- The JDO architecture is implementable for a wide range of heterogeneous local file systems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

3 JDO Architecture

3.1 Overview

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

Figure 2.0 Overview of non-managed JDO architecture



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

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

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

3.2 JDO Architecture

3.2.1 Two tier usage

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

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

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

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

3.2.2 Application server usage

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

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

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

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

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

Resource Adapter

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

The resource adapter plugs into a container (provided by the application server). The application components deployed on the container then use the client API exposed by `javax.jdo.PersistenceManager` to access the JDO `PersistenceManager`. The JDO implementation in turn uses the underlying resource adapter interface specific to the 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 `PersistenceManagers` might be pooled, and the underlying connections to the datastores might be independently pooled.

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

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

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

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

Contracts

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

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

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

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

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

Figure 3.0 Contracts between application server and native JDO resource adapter

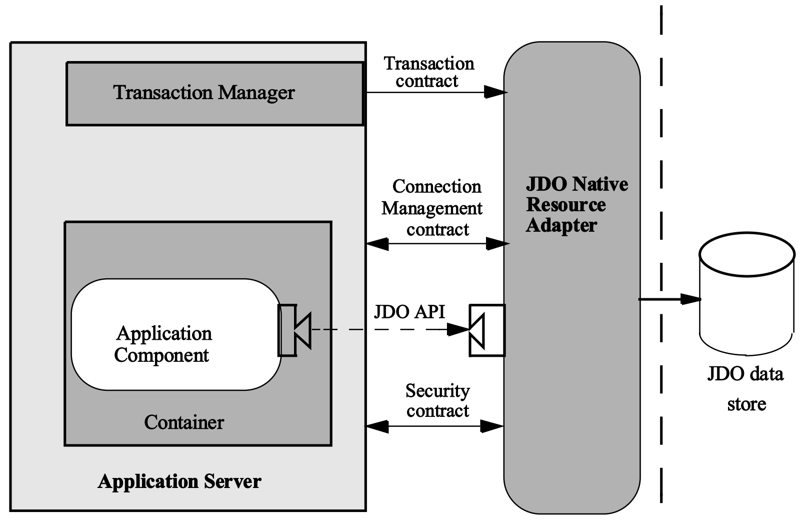
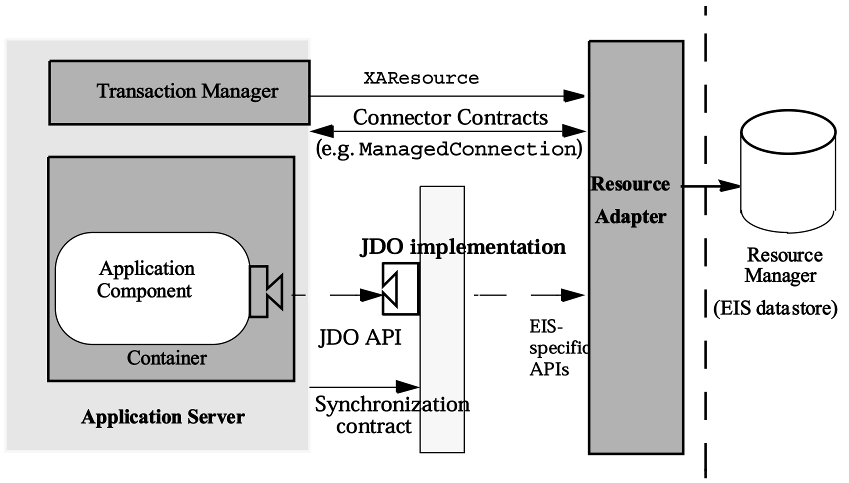


Figure 4.0 Contracts between application server and layered JDO implementation



5 Life Cycle of JDO Instances

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

5.1 Overview

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

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

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

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

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

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

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

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

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

5.2 Goals

The JDO instance life cycle has the following goals:

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

5.3 Architecture:

JDO Instances

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

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

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

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

JDO State Manager

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

JDO Managed Fields

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

5.4 JDO Identity

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

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

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

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

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

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

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

Three Types of JDO identity

JDO defines three types of JDO identity:

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

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

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

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

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

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

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

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

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

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

Uniquing

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

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

Change of identity

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

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

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

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

JDO Identity Support

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

5.4.1 Application (primary key) identity

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

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

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

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

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

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

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

5.4.2 Datastore identity

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

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

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

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

5.4.3 Nondurable JDO identity

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

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

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

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

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

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

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

5.5 Life Cycle States

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

Datastore Transactions

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

5.5.1 Transient (Required)

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

There is no JDO identity associated with a transient instance.

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

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

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

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

5.5.2 Persistent-new (Required)

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

During the transition from transient to persistent-new

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

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

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

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

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

5.5.3 Persistent-dirty (Required)

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

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

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

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

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

5.5.4 Hollow (Required)

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

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

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

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

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

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

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

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

5.5.5 Persistent-clean (Required)

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

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

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

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

5.5.6 Persistent-deleted (Required)

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

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

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

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

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

5.5.7 Persistent-new-deleted (Required)

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

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

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

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

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

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

5.6 Nontransactional (Optional)

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

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

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

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

With `NontransactionalRead` set to `true`:

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

With `NontransactionalWrite` set to `true`:

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

With `RetainValues` set to `true`:

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

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

With `RestoreValues` set to `true`:

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

5.6.1 Persistent-nontransactional (Optional)

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

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

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

As long as a transaction is not in progress:

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

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

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

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

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

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

5.7 Transient Transactional (Optional)

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

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

5.7.1 Transient-clean (Optional)

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

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

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

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

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

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

5.8 Optimistic Transactions (Optional)

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

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

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

Table 2: State Transitions

method \ current state	Transient	P-new	P-clean	P-dirty	Hollow
makePersistent	P-new	unchanged	unchanged	unchanged	unchanged
deletePersistent	error	P-new-del	P-del	P-del	P-del
makeTransactional	T-clean	unchanged	unchanged	unchanged	P-clean
makeNontransactional	error	error	P-nontrans	error	unchanged
makeTransient	unchanged	error	Transient	error	Transient
commit retainValues=false	unchanged	Hollow	Hollow	Hollow	unchanged
commit retainValues=true	unchanged	P-nontrans	P-nontrans	P-nontrans	unchanged
rollback restoreValues=false	unchanged	Transient	Hollow	Hollow	unchanged
rollback restoreValues=true	unchanged	Transient	P-nontrans	P-nontrans	unchanged
refresh with active Datastore transaction	unchanged	unchanged	unchanged	P-clean	unchanged
refresh with active 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

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

error: a `JDOUserException` is thrown; the state does not change
unchanged: no state change takes place; no exception is thrown due to the state change
n/a: not applicable; if this instance is an explicit parameter of the method, a `JDOUserException` is thrown; if this instance is an implicit parameter, it is ignored.
impossible: the state cannot occur in this scenario

Figure 7.0 Life Cycle: New Persistent Instances

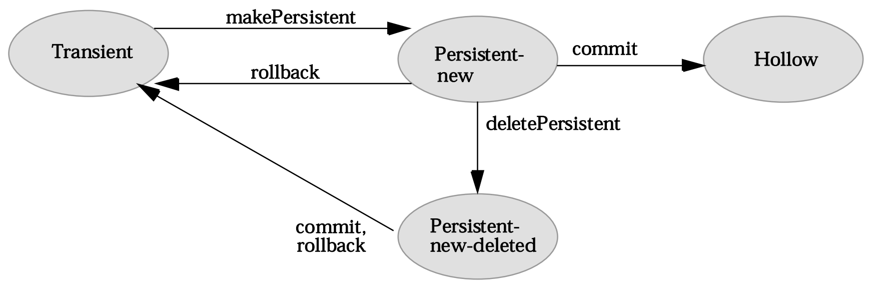


Figure 8.0 Life Cycle: Transactional Access

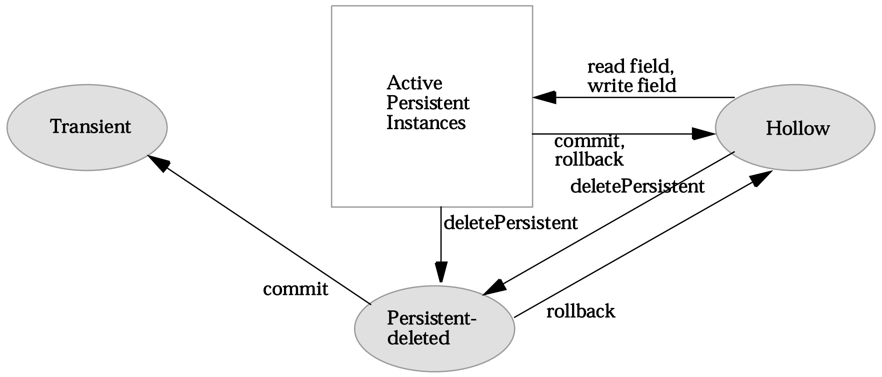


Figure 9.0 Life Cycle: Datastore Transactions

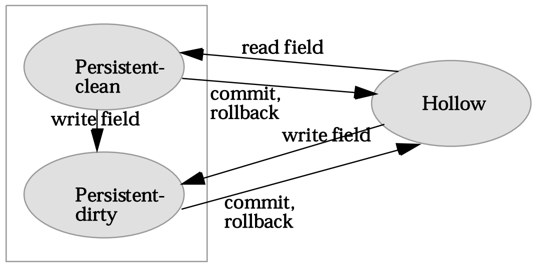


Figure 10.0 Life Cycle: Optimistic Transactions

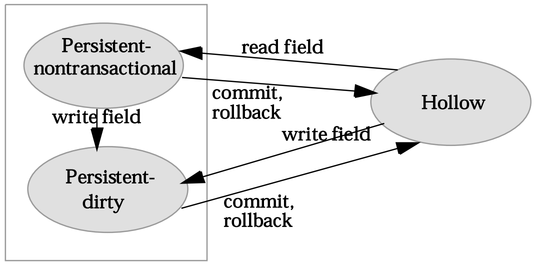


Figure 11.0 Life Cycle: Access Outside Transactions

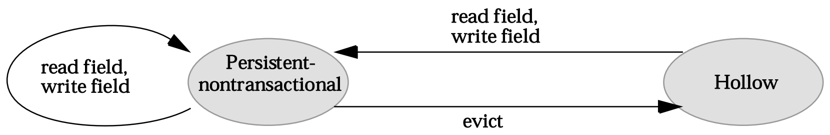


Figure 12.0 Life Cycle: Transient Transactional

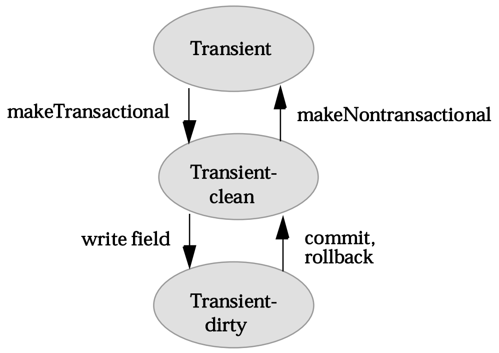
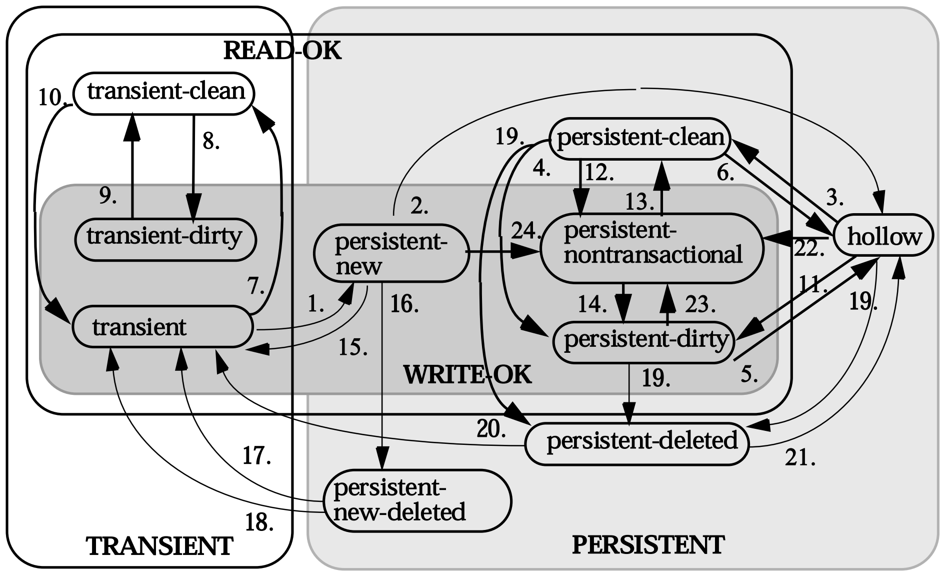


Figure 13.0 JDO Instance State Transitions



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

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

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

6 The Persistent Object Model

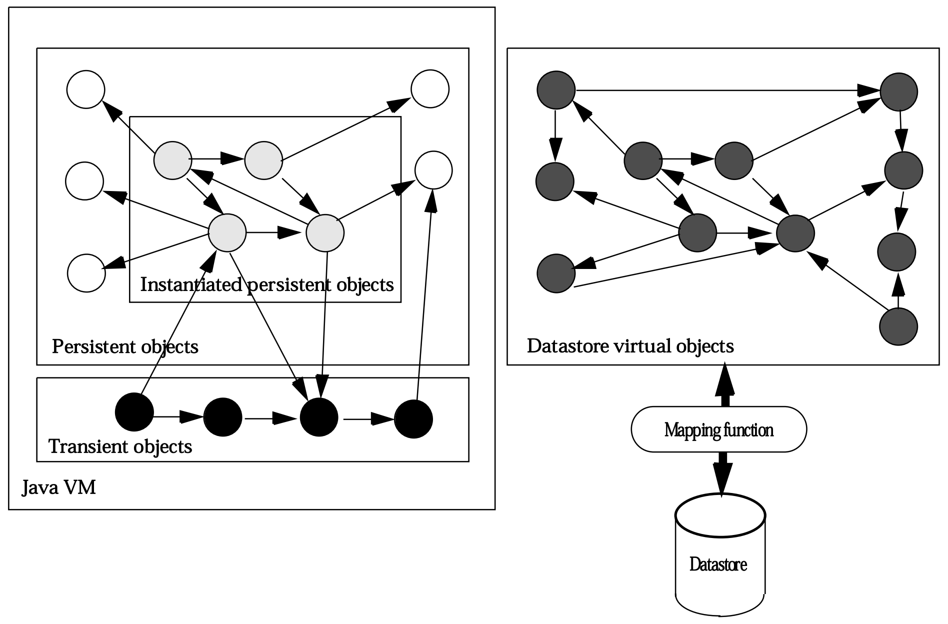
This chapter specifies the object model for persistence capable classes. To the extent possible, the object model is the same as the Java object model. Differences between the Java object model and the JDO object model are highlighted.

6.1 Overview

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

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

Figure 14.0 Instantiated persistent objects



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

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

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

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

6.2 Goals

The JDO Object Model has the following objectives:

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

6.3 Architecture

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

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

`PersistenceCapable` interface

The JDO Object Model distinguishes between two kinds of classes: those that implement `PersistenceCapable` and those that don't. A user-defined class can implement `PersistenceCapable` unless its state depends on the state of inaccessible or remote objects (e.g. it extends `java.net.SocketImpl` or uses JNI (native calls) to implement `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 `PersistenceCapable` that has JDO Identity and can be stored in a datastore, and independently deleted and queried. A Second Class Object (SCO) has no JDO Identity of its own and is stored in the datastore only as part of a First Class Object. In some JDO implementations, some SCO instances are actually artifacts that have no literal datastore representation at all, but are used only to represent relationships. For example, a `Collection` of a `PersistenceCapable` class might not be stored in the datastore, but created when needed to represent the relationship in memory. At commit time, the memory artifact is discarded and the relationship is represented entirely by datastore relationships.

First Class Objects

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

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

Second Class Objects

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

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

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

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

It is possible for an application to assign the same instance of a mutable SCO class to multiple FCO embedded fields, but this non-portable behavior is strongly discouraged for the following reason. If the assignment is done to persistent-new, persistent-clean, or persistent-dirty instances, then at the time that the FCOs are committed to the datastore, the Java object identity of the owned SCOs might change, because each FCO might have its own unshared SCO. If the assignment is done before `makePersistent` is called to make the FCOs persistent, the embedded fields are immediately replaced by copies, and no sharing takes place.

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

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

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

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

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

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

Arrays

Arrays are system-defined classes that do not necessarily have any JDO Identity of their own, and support by a JDO implementation is optional. If an implementation supports them, they might be stored in the datastore as part of an FCO. They do not support 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 `jdoMakeDirty` method of the `PersistenceCapable` interface (or `makeDirty` of the `JDOHelper` class), or by replacing the field value with its current value.

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

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

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

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

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

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

Primitives

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

Interfaces

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

6.4 Field types of persistence-capable classes

6.4.1 Nontransactional non-persistent fields

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

6.4.2 Transactional non-persistent fields

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

6.4.3 Persistent fields

Primitive types

JDO implementations must support fields of any of the primitive types

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

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

Immutable Object Class types

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

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

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

Mutable Object Class types

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

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

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

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

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

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

PersistenceCapable Class types

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

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

Object Class type

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

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

Collection Interface types

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

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

Other Interface types

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

Portable JDO applications must treat these fields as FCOs.

Arrays

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

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

6.5 Inheritance

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

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

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

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

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

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

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

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

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

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

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

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

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

14 Query

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

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

14.1 Overview

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

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

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

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

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

14.2 Goals

The JDO `Query` interface has the following goals:

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

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

14.3 Architecture: Query

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

There are three required elements in any query:

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

Other elements in queries include:

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

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

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

14.4 Namespaces in queries

The query namespace is modeled after methods in Java:

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

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

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

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

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

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

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

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

14.5 Query Factory in PersistenceManager interface

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

```
Query newQuery();
```

Construct an empty query instance.

`Query newQuery (Object query);`

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

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

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

`Query newQuery (Class cls);`

Construct a query instance with the candidate class specified.

`Query newQuery (Extent cln);`

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

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

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

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

Construct a query instance with the candidate class and filter specified.

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

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

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

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

14.6 Query Interface

```
package javax.jdo;
interface Query extends Serializable {
```

Persistence Manager

`PersistenceManager getPersistenceManager();`

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

Query element binding

The `Query` interface provides methods to bind required and other elements prior to execution.

All of these methods replace the previously set query element, by the parameter. [The methods are not additive.] For example, if multiple variables are needed in the query, all of them must be specified in the same call to `declareVariables`.

`void setClass (Class candidateClass);`

Bind the candidate class to the query instance.

`void setCandidates (Collection candidateCollection);`

Bind the candidate `Collection` to the query instance. If the user adds or removes elements from the `Collection` after this call, it is not determined whether the added/removed elements take part in the `Query`, or whether a `NoSuchElementException` is thrown during execution of the `Query`.

For portability, the elements in the collection must be persistent instances associated with the same `PersistenceManager` as the `Query` instance. An implementation might support transient instances in the collection. If persistent instances associated with another `PersistenceManager` are in the collection, `JDOUserException` is thrown during `execute()`.

If the candidates are not specified explicitly by `newQuery`, `setCandidates(Collection)`, or `setCandidates(Extent)`, then the candidate extent is the extent of instances of the candidate class in the datastore including subclasses. That is, the candidates are the result of `getPersistenceManager().getExtent(candidateClass, true)`.

`void setCandidates (Extent candidateExtent);`

Bind the candidate `Extent` to the query instance.

`void setFilter (String filter);`

Bind the query filter to the query instance.

`void declareImports (String imports);`

Bind the import statements to the query instance. All imports must be declared in the same method call, and the imports must be separated by semicolons.

`void declareVariables (String variables);`

Bind the variable statements to the query instance. This method defines the types and names of variables that will be used in the filter but not provided as values by the `execute` method.

`void declareParameters (String parameters);`

Bind the parameter statements to the query instance. This method defines the parameter types and names that will be used by a subsequent `execute` method.

`void setOrdering (String ordering);`

Bind the ordering statements to the query instance.

Query options

`void setIgnoreCache (boolean flag);`

`boolean getIgnoreCache ();`

The `IgnoreCache` option, when set to `true`, is a hint to the query engine that the user expects queries be optimized to return approximate results by ignoring changed values in the cache. This option is useful only for optimistic transactions and allows the datastore to return results that do not take modified cached instances into account. An implementation may choose to ignore the setting of this flag, and always return exact results reflecting current cached values, as if the value of the flag were `false`.

Query compilation

The `Query` interface provides a method to compile queries for subsequent execution.

```
void compile();
```

This method requires the `Query` instance to validate any elements bound to the query instance and report any inconsistencies by throwing a `JDOUserException`. It is a hint to the `Query` instance to prepare and optimize an execution plan for the query.

14.6.1 Query execution

The `Query` interface provides methods that execute the query based on the parameters given. They return an unmodifiable `Collection` which the user can iterate to get results. Executing any operation on the `Collection` that might change it throws `UnsupportedOperationException`. For future extension, the signature of the `execute` methods specifies that they return an `Object` that must be cast to `Collection` by the user.

Any parameters passed to the `execute` methods are used only for this execution, and are not remembered for future execution.

For portability, parameters of persistence-capable types must be persistent or transactional instances. Parameters that are persistent or transactional instances must be associated with the same `PersistenceManager` as the `Query` instance. An implementation might support transient instances of persistence-capable types as parameters. If a persistent instance associated with another `PersistenceManager` is passed as a parameter, `JDOUserException` is thrown during `execute()`.

Queries may be constructed at any time before the `PersistenceManager` is closed, but may be executed only at certain times. If the `PersistenceManager` that constructed the `Query` is closed, then the `execute` methods throw `JDOUserException`. If the `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 is an unmodifiable collection of instances that satisfy the boolean filter. The result may be a large `Collection`, which should be iterated or possibly passed to another `Query`. The `size()` method might return `Integer.MAX_VALUE` if the actual size of the result is not known (for example, the `Collection` represents a cursored result).

When using an `Extent` to define candidate instances, the contents of the extent are subject to the setting of the `ignoreCache` flag. With `ignoreCache` set to `false`:

- if instances were made persistent in the current transaction, the instances will be considered part of the candidate instances.
- if instances were deleted in the current transaction, the instances will not be considered part of the candidate instances.

With `ignoreCache` set to `true`:

- if instances were made persistent in the current transaction, the new instances might not be considered part of the candidate instances.
- if instances were deleted in the current transaction, the instances will not be considered part of the candidate instances.

Each parameter of the `execute` method(s) is an `Object` that is either the value of the corresponding parameter or the wrapped value of a primitive parameter. The parameters associate in order with the parameter declarations in the `Query` instance.

```
Object executeWithMap (Map m);
```

The `executeWithMap` method is similar to the `execute` method, but takes its parameters from a `Map` instance. The `Map` contains key/value pairs, in which the key is the declared parameter name, and the value is the value to use in the query for that parameter. Unlike `execute`, there is no limit on the number of parameters.

```
Object executeWithArray (Object[] a);
```

The `executeWithArray` method is similar to the `execute` method, but takes its parameters from an array instance. The array contains `Objects`, in which the positional `Object` is the value to use in the query for that parameter. Unlike `execute`, there is no limit on the number of parameters.

14.6.2 Filter specification

The filter specification is a `String` containing a boolean expression that is to be evaluated for each of the instances in the candidate collection. If the filter is not specified, then it defaults to `"true"`, and the input `Collection` is filtered only for class type.

An element of the candidate collection is returned in the result if:

- it is assignment compatible to the candidate `Class` of the `Query`; and
- for all variables there exists a value for which the filter expression evaluates to `true`. The user may denote uniqueness in the filter expression by explicitly declaring an expression (for example, `e1 != e2`). For example, a filter for a `Department` where there exists an `Employee` with more than one dependent and an `Employee` making more than `30,000` might be: `"(emps.contains(e1) & e1.dependents > 1) & (emps.contains(e2) & e2.salary > 30000)"`. The same `Employee` might satisfy both conditions. But if the query required that there be two different `Employees` satisfying the two conditions, an additional expression could be added: `"(emps.contains(e1) & e1.dependents > 1) & (emps.contains(e2) & (e2.salary > 30000 & e1 != e2))"`.

Rules for constructing valid expressions follow the Java language, except for these differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of `Date` fields and `Date` parameters are valid.
- Equality and ordering comparisons of `String` fields and `String` parameters are valid. The comparison is done according to an ordering not specified by JDO. This allows an implementation to order according to a datastore-specified ordering, which might be locale-specific.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.

- The assignment operators =, +=, etc. and pre- and post-increment and -decrement are not supported.
- Methods, including object construction, are not supported, except for `Collection.contains(Object o)`, `Collection.isEmpty()`, `String.startsWith(String s)`, and `String.endsWith(String e)`. Implementations might choose to support non-mutating method calls as non-standard extensions.
- Navigation through a null-valued field, which would throw `NullPointerException`, is treated as if the subexpression returned `false`. Similarly, a failed cast operation, which would throw `ClassCastException`, is treated as if the subexpression returned `false`. Other subexpressions or other values for variables might still qualify the candidate instance for inclusion in the result set.
- Navigation through multi-valued fields (`Collection` types) is specified using a variable declaration and the `Collection.contains(Object o)` method.
- The following literals are supported, as described in the Java Language Specification: `IntegerLiteral`, `FloatingPointLiteral`, `BooleanLiteral`, `CharacterLiteral`, `StringLiteral`, and `NullLiteral`.

Note that comparisons between floating point values are by nature inexact. Therefore, equality comparisons (== and !=) with floating point values should be used with caution.

Identifiers in the expression are considered to be in the name space of the specified class, with the addition of declared imports, parameters and variables. As in the Java language, `this` is a reserved word, and it refers to the element of the collection being evaluated.

Identifiers that are persistent field names are required to be supported by JDO implementations. Identifiers that are not persistent field names (including final and static field names) might be supported but are not required. Portable queries must not use non-persistent, final, or static field names in filter expressions.

Navigation through single-valued fields is specified by the Java language syntax of `field_name.field_name.....field_name`.

A JDO implementation is allowed to reorder the filter expression for optimization purposes.

The following are minimum capabilities of the expressions that every implementation must support:

- operators applied to all types where they are defined in the Java language:

Table 4: Query Operators

Operator	Description
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal

Table 4: Query Operators

Operator	Description
<=	less than or equal
&	boolean logical AND (not bitwise)
&&	conditional AND
|	boolean logical OR (not bitwise)
||	conditional OR
~	integral unary bitwise complement
+	binary or unary addition or String concatenation
-	binary subtraction or numeric sign inversion
*	times
/	divide by
!	logical complement

- exceptions to the above:
  - String concatenation is supported only for `String` + `String`, not `String` + `<primitive>`;
- parentheses to explicitly mark operator precedence
- cast operator (class)
- promotion of numeric operands for comparisons and arithmetic operations. The rules for promotion follow the Java rules (see chapter 5.6 Numeric Promotions of the Java language spec) extended by `BigDecimal`, `BigInteger` and numeric wrapper classes:
  - if either operand is of type `BigDecimal`, the other is converted to `BigDecimal`.
  - otherwise, if either operand is of type `BigInteger`, and the other type is a floating point type (`float`, `double`) or one of its wrapper classes (`Float`, `Double`) both operands are converted to `BigDecimal`.
  - otherwise, if either operand is of type `BigInteger`, the other is converted to `BigInteger`.
  - otherwise, if either operand is of type `double`, the other is converted to `double`.
  - otherwise, if either operand is of type `float`, the other is converted to `float`.
  - otherwise, if either operand is of type `long`, the other is converted to `long`.
  - otherwise, both operands are converted to type `int`.

  - operands of numeric wrapper classes are treated as their corresponding primitive types. If one of the operands is of a numeric wrapper class and the other operand is of a primitive numeric type, the rules above apply and the result is of the corresponding numeric wrapper class.
- equality comparison among persistent instances of `PersistenceCapable` types use the JDO Identity comparison of the references. Thus, two objects will compare equal if they have the same JDO Identity.
- comparisons between persistent and non-persistent instances return not equal.
- equality comparison of instances of non-`PersistenceCapable` reference types uses the `equals` method of the type.
- `String` methods `startsWith` and `endsWith` support wild card queries. JDO does not define any special semantic to the argument passed to the method; in particular, it does not define any wild card characters.
- `Null`-valued fields of `Collection` types are treated as if they were empty if a method is called on them. In particular, they return `true` to `isEmpty` and return `false` to all `contains` methods. For datastores that support `null` values for `Collection` types, it is valid to compare the field to `null`. Datastores that do not support `null` values for `Collection` types, will return `false` if the query compares the field to `null`. Datastores that support `null` values for `Collection` types should include the option `"javax.jdo.option.NullCollection"` in their list of supported options (`PersistenceManagerFactory.supportedOptions()`).

14.6.3 Parameter declaration

The parameter declaration is a `String` containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

Parameter types for primitive values can be specified as either the primitive types or the corresponding wrapper types. If a parameter type is specified as a primitive, the parameter value passed to `execute()` must not be null.

14.6.4 Import statements

The import statements follow the Java syntax for import statements.

14.6.5 Variable declaration

The type declarations follow the Java syntax for local variable declarations.

If the variable is not named in a contains clause, that variable's scope while evaluating the filter expression is the `Extent` (including subclasses) of the class of the variable. If the class does not manage an `Extent`, then no results will satisfy the query.

A portable query will constrain all variables with a `contains` clause in each "OR" expression of the filter where the variable is used. Further, the `contains` clause must be the left expression of an "AND" expression where the variable is used in the right expression. That is, for each occurrence of an expression in the filter using the variable, there is a `contains` clause "ANDed" with the expression that constrains the possible values by the elements of a collection.

A variable that is not constrained with an explicit `contains` clause is constrained by the extent of the persistence capable class in the database.

The semantics of contains is "exists". The meaning of the expression "emps.contains(e) && e.salary < param" is "there exists an e in the emps collection such that e.salary is less than

param". This is the natural meaning of contains in the Java language, except where the expression is negated.

If the expression is negated, then "!(emps.contains(e) && e.salary < param)" means "there does not exist an employee e in the collection emps such that e.salary is less than param". Another way of expressing this is "for each employee e in the collection emps, e.salary is greater than or equal to param".

14.6.6 Ordering statement

The ordering statement is a `String` containing one or more ordering declarations separated by commas. Each ordering declaration is a Java expression of an orderable type:

- primitives except `boolean`;
- wrappers except `Boolean`;
- `BigDecimal`;
- `BigInteger`;
- `String`;
- `Date`

followed by one of the following words: "`ascending`" or "`descending`".

Ordering might be specified including navigation. The name of the field to be used in ordering via navigation through single-valued fields is specified by the Java language syntax of `field_name.field_name....field_name`.

The result of the first (leftmost) expression is used to order the results. If the leftmost expression evaluates the same for two or more elements, then the second expression is used for ordering those elements. If the second expression evaluates the same, then the third expression is used, and so on until the last expression is evaluated. If all of the ordering expressions evaluate the same, then the ordering of those elements is unspecified.

The ordering of instances containing null-valued fields specified by the ordering is not specified. Different JDO implementations might order the instances containing null-valued fields either before or after instances whose fields contain non-null values.

14.6.7 Closing Query results

When the application has finished with the query results, it might optionally close the results, allowing the JDO implementation to release resources that might be engaged, such as database cursors or iterators. The following methods allow early release of these resources.

```
void close (Object queryResult);
```

This method closes the result of one `execute(...)` method, and releases resources associated with it. After this method completes, the query result can no longer be used, for example to iterate the returned elements. Any elements returned previously by iteration of the results remain in their current state. Any iterators acquired from the queryResult will return `false` to `hasNext()` and will throw `NoSuchElementException` to `next()`.

```
void closeAll ();
```

This method closes all results of `execute(...)` methods on this `Query` instance, as above. The `Query` instance is still valid and can still be used.

14.7 Examples:

The following class definitions for persistence capable classes are used in the examples:

```
package com.xyz.hr;
class Employee {
String name;
Float salary;
Department dept;
Employee boss;
}
package com.xyz.hr;
class Department {
String name;
Collection emps;
}
```

14.7.1 Basic query.

This query selects all Employee instances from the candidate collection where the salary is greater than the constant `30000`.

Note that the `float` value for `salary` is unwrapped for the comparison with the literal `int` value, which is promoted to `float` using numeric promotion. If the value for the `salary` field in a candidate instance is `null`, then it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

14.7.2 Basic query with ordering.

This query selects all Employee instances from the candidate collection where the salary is greater than the constant 30000, and returns a `Collection` ordered based on employee salary.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000";
Query q = pm.newQuery (clnEmployee, filter);
q.setOrdering ("salary ascending");
Collection emps = (Collection) q.execute ();
```

14.7.3 Parameter passing.

This query selects all Employee instances from the candidate collection where the salary is greater than the value passed as a parameter.

If the value for the `salary` field in a candidate instance is `null`, then it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > sal";
Query q = pm.newQuery (clnEmployee, filter);
String param = "Float sal";
q.declareParameters (param);
Collection emps = (Collection) q.execute (new Float (30000.));
```

14.7.4 Navigation through single-valued field.

This query selects all Employee instances from the candidate collection where the value of the name field in the Department instance associated with the Employee instance is equal to the value passed as a parameter.

If the value for the `dept` field in a candidate instance is `null`, then it cannot be navigated for the comparison, and the candidate instance is rejected.

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "dept.name == dep";
String param = "String dep";
Query q = pm.newQuery (clnEmployee, filter);
q.declareParameters (param);
String rnd = "R&D";
Collection emps = (Collection) q.execute (rnd);
```

14.7.5 Navigation through multi-valued field.

This query selects all `Department` instances from the candidate collection where the collection of `Employee` instances contains at least one Employee instance having a salary greater than the value passed as a parameter.

```
Class depClass = Department.class;
Extent clnDepartment = pm.getExtent (depClass, false);
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
String param = "float sal";
Query q = pm.newQuery (clnDepartment, filter);
q.declareParameters (param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

14.7.6 Membership in a collection

This query selects all `Department` instances where the name field is contained in a parameter collection, which in this example consists of three department names.

```
Class depClass = Department.class;
Extent clnDepartment = pm.getExtent (depClass, false);
String filter = "depts.contains(name)";
List depts =
   Arrays.asList(new String [] {"R&D", "Sales", "Marketing"};
String param = "Collection depts";
Query q = pm.newQuery (clnDepartment, filter);
q.declareParameters (param);
Collection deps = (Collection) q.execute (depts);
```

24 Items deferred to the next release

This chapter contains the list of items that were raised during the development of JDO but were not resolved.

24.1 Nested Transactions

Define the semantics of nested transactions.

24.2 Savepoint, Undosavepoint

Related to nested transactions, savepoints allow for making changes to instances and then undoing those changes without making any datastore changes. It is a single-child nested transaction.

24.3 Inter-PersistenceManager References

Explain how to establish and maintain relationships between persistent instances managed by different `PersistenceManagers`.

24.4 Enhancer Invocation API

A standard interface to call the enhancer will be defined.

24.5 Prefetch API

A standard interface to specify prefetching of instances by policy will be defined. The intended use it to allow the application to specify a policy whereby instances of persistence capable classes will be prefetched from the datastore when related instances are fetched. This should result in improved performance characteristics if the prefetch policy matches actual application access patterns.

24.6 BLOB/CLOB datatype support

JDO implementations can choose to implement mapping from java.sql.Blob datatype to byte arrays, and java.sql.Clob to String or other java type; but these mappings are not standard, and may not have the performance characteristics desired.

24.7 Managed (inverse) relationship support

In order for JDO implementations to be used for container managed persistence entity beans, relationships among persistent instances need to be explicitly managed. See the EJB Specification 2.0, sections 9.4.6 and 9.4.7 for requirements. The intent is to support these semantics when the relationships are identified in the metadata as inverse relationships.

24.8 Case-Insensitive Query

Use of String.toLowerCase() as a supported method in query filters would allow case-insensitive queries.

24.9 String conversion in Query

Supported String constructors String(<integer expression>) and String(<floating-point expression>) would make queries more flexible.

24.10 Read-only fields

Support (probably marking the fields in the XML metadata) for read-only fields would allow better support for databases where modification of data elements is proscribed. The metadata annotation would permit earlier detection of incorrect modification of the corresponding fields.

24.11 Enumeration pattern

The enumeration pattern is a powerful technique for emulating enums. The pattern in summary allows for fields to be declared as:

```
class Foo {
   Bar myBar = Bar.ONE;
   Bar someBar = new Bar("illegal"); // doesn't compile
}
class Bar {
   private String istr;
   private Bar(String s) {
      istr = s;
   }
   public static Bar ONE = new Bar("one");
   public static Bar TWO = new Bar("two");
}
```

The advantage of this pattern is that fields intended to contain only certain values can be constrained to those values. Supporting this pattern explicitly allows for classes that use this pattern to be supported as persistence-capable classes.

24.12 Non-static inner classes

Allow non-static inner classes to be persistence-capable. The implication is that the enclosing class must also be persistence-capable, and there is a one-many relationship between the enclosing class and the inner class.

24.13 Projections in query

Currently the only return value from a JDOQL query is a Collection of persistent instances. Many applications need values returned from queries, not instances. For example, to properly support EJBQL, projections are required. One way to provide projections is to model what EJBQL has already done, and add a method setResult (String projection) to javax.jdo.Query. This method would take as a parameter a single-valued navigation expression. The result of execute for the query would be a Collection of instances of the expression.

24.14 LogWriter support

Currently, there is no direct support for writing log messages from an implementation, although there is a connection factory property that can be used for this purpose. A future revision could define how an implementation should use a log writer.

24.15 New Exceptions

Some exceptions might be added to more clearly define the cause of an exception. Candidates include `JDODuplicateObjectIdException`, `JDOClassNotPersistenceCapableException`, `JDOExtentNotManagedException`, `JDOConcurrentModificationException`, `JDOQueryException`, `JDOQuerySyntaxException`, `JDOUnboundQueryParameterException`, `JDOTransactionNotActiveException`, `JDODeletedObjectFieldAccessException`.

24.16 Distributed object support

Provide for remote object graph support, including instance reconciliation, relationship graph management, instance insertion ordering, etc.

24.17 Object-Relational Mapping

Extend the current xml metadata to include optional O/R mapping information. This could include tables to map to classes, columns to map to fields, and foreign keys to map to relationships.

Other O/R mapping issues include sequence generation for primary key support.

Appendix B: Design Decisions

This appendix outlines some of the design decisions that were considered and not taken, along with the rationale.

B.1 Enhancer

The enhancer could generate code that would delegate to the associated StateManager every access (read or write) for every field. This design was rejected because of several factors.

- Code bloat: the enhanced code would add an extra method call to every access to a persistent field.
- Performance: the calls to the `StateManager` would add extra cycles to every access to a persistent field, even if the field were already fetched into the persistent instance.

The enhancer could require complete metadata descriptions for all persistence-capable classes and persistent and transactional fields, and further require that all classes be available during enhancement of any class.

This would allow the enhancer to generate the most efficient code, but imposes an extra burden on the user to keep the metadata and class definition absolutely in sync. If a field were declared in a class after the metadata was defined, the user would have to update the metadata to add the new field.

Requiring access to all classes during enhancement of any class was also seen as an extra burden on the user, who would have to execute the enhancement in an environment that did not necessarily reflect the runtime environment. There is also a performance penalty and additional complexity for the enhancer.

The decision that was taken was that the enhancer must be able to determine the persistence-modifier (persistent or none) from the Java modifiers and type of a field. Further, the information needed to enhance a class is only the class file for the class being enhanced, plus the metadata for the class and classes directly reachable (via references or inheritance) from the class.

The java byte codes generated in a class for a field in another class do not contain much information about the modifiers (final or transient) of the field. They do have the field name and the field type, and whether the field is static. There is an implied access control that permits the generated access (package, protected, or public) but no distinction among the choices.

Therefore, a field that is not declared in the metadata must be enhanced to generate an accessor and mutator even though the field is not persistent. For example, for a final int field declared in a class, the field is not persistent, so it is not included in the list of persistent/transactional fields, but an accessor is generated for it. This accessor will be used only by other classes' accesses, and access will not be mediated (the StateManager will never be called). Accesses within the class are not enhanced.

B.2 PersistenceCapable

The PersistenceCapable interface could be eliminated entirely in favor of having all interrogatives operate via the PersistenceManager, not directly on the JDO instance. This would make the JDO instance entirely user-written. However, the impact would be that to find out which PersistenceManager, if any, was responsible for the JDO instance, a new

singleton would have to be provided. The singleton would have to register all PersistenceManager instances and ask each if it managed a specific JDO instance.

This was deemed too complex to manage, as well as too slow to find simple information that should be easily available.

B.3 Collection Factory

The collection factory could be defined as methods on PersistenceManager or as methods on a separate interface. Also, a single method that takes a type, or multiple methods, one for each type could be defined.

The decision was taken to define two methods on PersistenceManager based on the requirement to create an instance of a collection based on the type of an existing instance. This operation would be complex if individual methods were used, one per type.

A convenience interface can easily be created using the defined methods.