ObjectDB 2.3 Developer's Guide

Last updated on September 21, 2011.

Table of Contents

Preface

Welcome to ObjectDB for Java/JPA Developer's Guide. Here you can learn how to develop database applications using ObjectDB and JPA (Java Persistence API). The main purpose of this guide is to make you productive with ObjectDB and JPA in a short time.

Organization of this Guide

This manual is divided into the following six chapters:

- Chapter 1 - Quick Tour

Demonstrates basic database programming using ObjectDB and JPA.

- Chapter 2 - JPA Entity Classes

Shows how to define JPA entity classes that can be persisted in ObjectDB.

- Chapter 3 - Using JPA

Shows how to use JPA to store, retrieve, update and delete database objects.

- Chapter 4 - JPA Queries (JPQL / Criteria)

Explains how to use the JPA Query Language (JPQL).

- Chapter 5 - Database Tools and Utilities

Presents ObjectDB Tools: the Explorer, the Enhancer, the Doctor, etc.

- Chapter 6 - Configuration

Describes the ObjectDB configuration and explains how to tune ObjectDB.

Prerequisite Knowledge

A prior knowledge of database programming (SQL, JDBC, ORM or JPA) is not required in order to follow this guide, but a strong background and understanding of the Java language is essential.

Additional Reading and Resources

This guide focuses mainly on practical issues in order to make the reader productive in a short time. After reading this guide you may want to extend your knowledge of JPA, by reading a book on JPA.

Feedback

We would appreciate any comment or suggestion regarding this manual.
Please send your comments or questions to support@objectdb.com .

Chapter 1 - Quick Tour

This chapter demonstrates basic ObjectDB and JPA concepts by introducing a simple example program. After reading this chapter you should be able to write basic programs that create, open and close ObjectDB databases and perform basic CRUD operations (Create/Store, Retrieve, Update and Delete) on ObjectDB databases.

The example program that this chapter presents manages a simple database that contains points in the plane. Each point is represented by an object with two `int` fields, x and y, that hold the point's x and y coordinates. The program demonstrates CRUD database operations by storing, retrieving, updating and deleting `Point` objects.

This chapter contains the following sections:

To run the sample program of this chapter in your IDE refer to one of the following tutorials:

- Getting Started with JPA and Eclipse Tutorial
- Getting Started with JPA and NetBeans Tutorial

These tutorials provide step by step instructions on how to start using JPA in your IDE with the ObjectDB object database. Given the simplicity of ObjectDB, that should be quick and easy even for a novice.

1.1 Defining a JPA Entity Class

To be able to store `Point` objects in the database using JPA we need to define *an entity class*. A JPA entity class is a POJO (Plain Old Java Object) class, i.e. an ordinary Java class that is marked (annotated) as having the ability to represent objects in the database. Conceptually this is similar to serializable classes, which are marked as having the ability to be serialized.

The Point Entity Class

The following `Point` class, which represents points in the plane, is marked as an entity class, and accordingly, provides the ability to store `Point` objects in the database and retrieve `Point` objects from the database:

```
package com.objectdb.tutorial;

@Entity
public class Point {
    // Persistent Fields:
    private int x;
    private int y;

    // Constructor:
    Point (int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Accessor Methods:
    public int getX() { return this.x; }
    public int getY() { return this.y; }

    // String Representation:
    @Override
    public String toString() {
        return String.format("(%d, %d)", this.x, this.y);
    }
}
```

As you can see above, an entity class is an ordinary Java class. The only unique JPA addition is the `@Entity` annotation, which marks the class as an entity class.

An attempt to persist `Point` objects in the database without marking the `Point` class as an entity class will cause a `PersistenceException`. This is similar to the `NotSerializableException` that Java throws (unrelated to JPA), on an attempt to serialize non serializable objects (except that all JPA exceptions are unchecked where Java IO exceptions are checked).

Persistent Fields in Entity Classes

Storing an entity object in the database does not store methods and code. Only the persistent state of the entity object, as reflected by its *persistent fields* is stored. By default, any field that is not declared as `static` or `transient` is a persistent field. In our example, the persistent fields of the `Point` entity class are `x` and `y` (representing the position of the point in the plane). It is the values of these fields that are stored in the database when an entity object is persisted.

Chapter 2 provides additional information on how to define entity classes, including which persistent types

can be used for persistent fields, how to define and use a primary key and what a version field is and how to use it.

If you are already familiar with JPA you might have noticed that the `Point` entity class has no primary key (`@Id`) field defined. As an object database, ObjectDB supports implicit object IDs, so an explicitly defined primary key is not required. On the other hand, ObjectDB also supports explicit JPA primary keys, including composite primary keys and automatic sequential value generation. This is a very powerful feature of ObjectDB that is absent from other object oriented databases.

1.2 Obtaining a JPA Database Connection

In JPA a database connection is represented by the `EntityManager` interface. Therefore, in order to manipulate an ObjectDB database we need an `EntityManager` instance. Operations that modify database content also require an `EntityTransaction` instance.

Obtaining an EntityManagerFactory

Obtaining an `EntityManager` instance consists of two steps. First we need to obtain an instance of `EntityManagerFactory` that represents the relevant database and then we can use that factory instance to get an `EntityManager` instance.

JPA requires the definition of a persistence unit in an XML file in order to be able to generate an `EntityManagerFactory`. But when using ObjectDB you can either define a standard persistence unit in an XML file or you can simply provide the file path of the ObjectDB database instead:

```
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("$objectdb/db/points.odb");
```

The `createEntityManagerFactory` static method expects a persistence unit name as an argument, but when using ObjectDB, any valid database file path (absolute or relative) is also accepted. Any string that ends with `.odb` or `.objectdb` is considered by ObjectDB to be a database url rather than a persistence unit name.
The `$objectdb` special prefix represents the ObjectDB home directory (by default - the directory in which ObjectDB is installed). If no database file exists yet at the given path ObjectDB will try to create one.

The `EntityManagerFactory` is also used to close the database once we are finished using it:

```
emf.close();
```

Obtaining an EntityManager

Once we have an EntityManagerFactory we can easily obtain an EntityManager instance:

```
EntityManager em = emf.createEntityManager();
```

The EntityManager instance represents a connection to the database. When using JPA, every operation on a database is associated with an EntityManager. Further, in a multithreaded application every thread usually has its own EntityManager instance while at the same time sharing a single application-wide EntityManagerFactory.

When the connection to the database is no longer needed the EntityManager can be closed:

```
em.close();
```

Closing an EntityManager does not close the database itself (that is the job of the factory as previously explained). Once the EntityManager object is closed it cannot be reused. But, the owning EntityManagerFactory instance may preserve the EntityManager's resources (such as a database file pointer or a socket to a remote server) in a connection pool and use them to speed up future EntityManager construction.

Using an EntityTransaction

Operations that modify database content, such as store, update, and delete should only be performed within an active transaction.

Given an EntityManager, em, it is very easy to begin a transaction:

```
em.getTransaction().begin();
```

There is a one to one relationship between an EntityManager instance and its associated EntityTransaction instances that the getTransaction method returns.

When a transaction is active you can invoke EntityManager methods that modify the database content, such as persist and remove. Database updates are collected and managed in memory and applied to the database when the transaction is committed:

```
em.getTransaction().commit();
```

The next section explains in more detail how to use the `EntityManager` and transactions for CRUD database operations.

1.3 CRUD Database Operations with JPA

Given an `EntityManager`, `em`, that represents a JPA connection to the object database, we can use it to store, retrieve, update and delete database objects.

Storing New Entity Objects

The following code fragment stores 1,000 `Point` objects in the database:

```
em.getTransaction().begin();
for (int i = 0; i < 1000; i++) {
    Point p = new Point(i, i);
    em.persist(p);
}
em.getTransaction().commit();
```

Operations that modify the content of the database (such as storing new objects) require an active transaction. In the example above, every `Point` object is first constructed as an ordinary Java object. It then becomes associated with an `EntityManager` and with its transaction (as a managed entity) by the `persist` method. The new `Point` objects are physically stored in the database only when the transaction is committed. The Storing Entities section in chapter 3 discusses persisting objects in the database in more detail.

JPA Queries with JPQL

We can get the number of `Point` objects in the database by using a simple query:

```
Query q1 = em.createQuery("SELECT COUNT(p) FROM Point p");
System.out.println("Total Points: " + q1.getSingleResult());
```

The query string ("SELECT COUNT(p) FROM Point p") instructs JPA to count all the Point objects in the database. If you have used SQL you should find the syntax very familiar. JPQL, the JPA query language,

supports a SQL like syntax. This has a couple of significant advantages. First, you get the power of SQL combined with the ease of use of object databases and second, a new JPA developer with some experience with SQL can become productive very quickly.

The `EntityManager` object serves as the factory for `Query` instances. The `getSingleResult` method executes the query and returns the result. It should only be used when exactly one result value is expected (a single Long object in the query above).

Let's see another example of a query that returns a single result:

```
Query q2 = em.createQuery("SELECT AVG(p.x) FROM Point p");
System.out.println("Average X: " + q2.getSingleResult());
```

The new query returns a Double object reflecting the average `x` value of all the `Point` objects in the database.

Retrieving Existing Entities

The Retrieving Entities section in chapter 3 describes several methods for retrieving entity objects from the database using JPA. Running a JPQL query is one of them:

```
TypedQuery<Point> query =
  em.createQuery("SELECT p FROM Point p", Point.class);
List<Point> results = query.getResultList();
```

The above query retrieves all the `Point` objects from the database as a list. The `TypedQuery` interface, which was introduced in JPA 2, is a type safe subinterface of `Query` and is usually the preferred way to work with queries. The `getResultList` method executes the query and returns the result objects. It should be used (rather than `getSingleResult`) when multiple results are expected. The result list can be iterated as any ordinary Java collection.

More advanced queries, for instance, can be used to retrieve selected objects from the database (using a WHERE clause), sort the results (using an ORDER BY clause) and even group results (using GROUP BY and HAVING clauses). JPQL is a very powerful query language and chapter 4 of this manual describes it in detail.

Updating and Deleting Entities

JPA refers to entity objects that are associated with an `EntityManager` as 'managed'. A newly constructed entity object becomes managed by an `EntityManager` when the `persist` method is invoked. Objects that

are retrieved from the database are managed by the `EntityManager` that was used to retrieve them (e.g. as a `Query` factory).

To delete an object from the database, you need to obtain a managed object (usually by retrieval) and invoke the `remove` method within the context of an active transaction:

```
em.getTransaction().begin();
em.remove(p); // delete entity
em.getTransaction().commit();
```

In the above code, `p` must be a managed entity object of the `EntityManager` `em`. The entity object is marked for deletion by the `remove` method and is physically deleted from the database when the transaction is committed.

Updating an existing database object is similar. You have to obtain a managed entity object (e.g. by retrieval) and modify it within an active transaction:

```
em.getTransaction().begin();
p.setX(p.getX() + 100); // update entity
em.getTransaction().commit();
```

You may notice that `em`, the managing EntityManager of `p`, is not mentioned explicitly when `p` is being updated. The `EntityManager` that manages an entity is responsible for automatically detecting changes to the entity object and applying them to the database when the transaction is committed.

The following code demonstrates processing of all the `Point` objects in the database:

```
TypedQuery<Point> query =
  em.createQuery("SELECT p FROM Point p", Point.class);
List<Point> results = query.getResultList();

em.getTransaction().begin();
for (Point p : results) {
    if (p.getX() >= 100) {
        em.remove(p); // delete entity
    }
    else {
        p.setX(p.getX() + 100); // update entity
    }
}
```

```
em.getTransaction().commit();
```

In the above example all the Point objects whose x coordinate is greater or equal to 100 are deleted. All the other Point objects are updated.

Chapter 3 of this manual describes how to use JPA for database operations in more detail.

1.4 What's next?

This chapter introduced the basic principles of using JPA with ObjectDB. You can dive into the details by reading the other chapters of this manual. If you prefer to get started using ObjectDB right away you can follow one of these tutorials to create and run the example program that was described in this chapter.

- Getting Started with JPA and Eclipse Tutorial
- Getting Started with JPA and NetBeans Tutorial

These tutorials explain how to run the sample program. You can easily start your own ObjectDB/JPA programs simply by modifying this sample program.

Reading the Next Chapters

The next three chapters provide more details on using JPA with ObjectDB:

- Chapter 2 - JPA Entity Classes
- Chapter 3 - Using JPA
- Chapter 4 - JPA Queries (JPQL / Criteria)

The last two chapters complete the picture by describing some tools and settings that are specific to ObjectDB:

- Chapter 5 - Database Tools and Utilities
- Chapter 6 - Configuration

Chapter 2 - JPA Entity Classes

JPA Entity classes are user defined classes whose instances can be stored in a database.
To store data in an ObjectDB database using JPA you have to define entity classes that represent your application data object model. This chapter explains how to define and use entity classes.

This chapter contains the following sections:

2.1 JPA Persistable Types

The term persistable types refers to data types that can be used in storing data in the database. ObjectDB supports all the JPA persistable types, which are:

- User defined classes - Entity classes, Mapped superclasses, Embeddable classes.
- Simple Java data types: Primitive types, Wrappers, String, Date and Math types.
- Multi value types - Collections, Maps and Arrays.
- Miscellaneous types: Enum types and Serializable types (user or system defined).

Note: Only instances of entity classes can be stored in the database directly. Other persistable types can be embedded in entity classes as fields. In addition, only instances of entity classes preserve identity and are stored only once even if they are referenced multiple times. Referencing instances of other persistable types from multiple persistent fields would cause data duplication in the database.

Entity Classes

An entity class is an ordinary user defined Java class whose instances can be stored in the database. The easy way to declare a class as an entity is to mark it with the Entity annotation:

```
import javax.persistence.Entity;

@Entity
```

```
public class MyEntity {

}
```

Entity Class Requirements

A portable JPA entity class:

- should be a top-level class (i.e. not a nested / inner class).
- should have a `public` or `protected` no-arg constructor.
- cannot be `final` and cannot have `final` methods or `final` instance variables.

ObjectDB is slightly less restrictive:

- Static nested entity classes are allowed (non static inner classes are forbidden).
- Instance (non static) variables cannot be final, but classes and methods can be final.
- In most cases ObjectDB can overcome a missing no-arg constructor.

Aside from these constraints an entity class is like any other Java class. It can extend either another entity class or a non-entity user defined class (but not system classes, such as `ArrayList`) and implement any interface. It can contain constructors, methods, fields and nested types with any access modifiers (public, protected, package or private) and it can be either concrete or abstract.

Entity Class Names

Entity classes are represented in queries by *entity names*. By default, the entity name is the unqualified name of the entity class (i.e. the short class name excluding the package name). A different entity name can be set explicitly by using the `name` attribute of the Entity annotation:

```
@Entity(name="MyName")
public class MyEntity {

}
```

Entity names must be unique. When two entity classes in different packages share the same class name, explicit entity name setting is required to avoid collision.

Mapped Superclasses

In JPA, classes that are declared as mapped superclasses have some of the features of entity classes, but

also some restrictions. ObjectDB, however, does not enforce these restrictions so mapped superclasses are treated by ObjectDB as ordinary entity classes.

Mapped superclasses are really only useful in applications that use an ORM-based JPA provider (such as Hibernate, TopLink, EclipseLink, OpenJPA, JPOX, DataNucleus, etc.).

Embeddable Classes

Embeddable classes are user defined persistable classes that function as value types. As with other non entity types, instances of an embeddable class can only be stored in the database as embedded objects, i.e. as part of a containing entity object.

A class is declared as embeddable by marking it with the Embeddable annotation:

```
@Embeddable
public class Address {
    String street;
    String city;
    String state;
    String country;
    String zip;
}
```

Instances of embeddable classes are always embedded in other entity objects and do not require separate space allocation and separate store and retrieval operations. Therefore, using embeddable classes can save space in the database and improve efficiency.

Embeddable classes, however, do not have an identity (primary key) of their own which leads to some limitations (e.g. their instances cannot be shared by different entity objects? and they cannot be queried directly), so a decision whether to declare a class as an entity or embeddable requires case by case consideration.

Simple Java Data Types

All the following simple Java data types are persistable:

- Primitive types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` and `double`.
- Equivalent wrapper classes from package java.lang:
  `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` and `Double`.
- `java.math.BigInteger`, `java.math.BigDecimal`.
- `java.lang.String`.

- `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`.

Date and time types are discussed in more detail in the next paragraph.

Date and Time (Temporal) Types

The `java.sql` date and time classes represent different parts of dates and times:

- `java.sql.Date` - represents date only (e.g. `2010-12-31`).
- `java.sql.Time` - represents time only (e.g. `23:59:59`).
- `java.sql.Timestamp` - represents date and time (e.g. `2010-12-31 23:59:59`).

The `java.util.Date` and `java.util.Calendar` types, on the other hand, are generic and can represent any of the above, using the @Temporal JPA annotation:

```
@Entity
public class DatesAndTimes {
    // Date Only:
    java.sql.Date date1;
    @Temporal(TemporalType.DATE) java.util.Date date2
    @Temporal(TemporalType.DATE) java.util.Calendar date3;

    // Time Only:
    java.sql.Time time1;
    @Temporal(TemporalType.TIME) java.util.Date time2;
    @Temporal(TemporalType.TIME) java.util.Calendar time3;

    // Date and Time:
    java.sql.Timestamp dateAndTime1
    @Temporal(TemporalType.TIMESTAMP) java.util.Date dateAndTime2;
    @Temporal(TemporalType.TIMESTAMP) java.util.Calendar dateAndTime3;
    java.util.Date dateAndTime4; // date and time but not JPA portable
    java.util.Calendar dateAndTime5; // date and time but not JPA portable
}
```

Persisting pure dates (without the time part), either by using the `java.sql.Date` type or by specifying the `@Temporal(TemporalType.DATE)` annotation has several benefits:

- It saves space in the database.
- It is more efficient (storage and retrieval is faster).
- It simplifies queries on dates and ranges of dates.

When an entity is stored, its date and time fields are automatically adjusted to the requested mode. For example, fields `date1`, `date2` and `date3` above may be initialized as `new Date()`, i.e. with both date and time. Their time part is discarded when they are stored in the database.

Multi Value Types

The following multi value types are persistable:

- Collection types from package `java.util`: `ArrayList`, `Vector`, `Stack`, `LinkedList`, `ArrayDeque`, `PriorityQueue`, `HashSet`, `LinkedHashSet`, `TreeSet`.
- Map types from package `java.util`: `HashMap`, `Hashtable`, `WeakHashMap`, `IdentityHashMap`, `LinkedHashMap`, `TreeMap` and `Properties`.
- Arrays (including multi dimensional arrays).

Both generic (e.g. `ArrayList<String>`) and non generic (e.g. `ArrayList`) collection and map types are supported, as long as their values (i.e. elements in collections and arrays and keys and values in maps) are either `null` values or instances of persistable types.

In addition to the collection and map classes that are fully supported by ObjectDB, any other class that implements `java.util.Collection` or `java.util.Map` can be used when storing entities. If an unsupported collection or map type is used, ObjectDB will switch to a similar supported type when the data is retrieved from the database.

For example, the `Arrays.asList` method returns an instance of an internal Java collection type that is not supported by ObjectDB. Nevertheless, the following entity can be stored:

```
@Entity
public class EntityWithList {
    private List<String> words = Arrays.asList("not", "ArrayList");
}
```

When the entity is retrieved from the database, the list will be instantiated as an `ArrayList`. Using an interface (`List<String>`) for the field type is essential in this case to enable switching to the supported collection type when the entity is retrieved. Actually, JPA requires declaring persistent collection and map fields only as interface types (i.e. `java.util.Collection`, `java.util.List`, `java.util.Set`, `java.util.Map`), and that is also a good practice when working with ObjectDB.

Proxy Classes

When entities are retrieved from the database, instances of mutable persistable system types (i.e.

collections, maps and dates) are instantiated using proxy classes that extend the original classes and enable transparent activation and transparent persistence.

For example, a collection that is stored as an instance of `java.util.ArrayList` is retrieved from the database as an instance of `objectdb.java.util.ArrayList`, which is a subclass of the original `java.util.ArrayList`.

Most applications are not affected by this, because proxy classes extend the original Java classes and inherit their behavior. The difference can be noticed in the debugger view and when invoking the `getClass` method on an object of such proxy classes.

Enum Types

Every enum type (user defined or system defined) is persistable. But, if future portability to other platforms is important, only values of user defined enum types should be persisted.

By default, enum values are represented internally by their ordinal numbers. Caution is required when modifying an enum type that is already in use in an exiting database. New enum fields can be added safely only at the end (with new higher ordinal numbers).

Alternatively, enum values can be represented internally by their names. In that case, the names must be fixed, since changing a name can cause data loss in existing databases.

The `@Enumerated` annotation enables choosing the internal representation:

```
@Entity
public class Style {
    Color color1; // default is EnumType.ORDINAL
    @Enumerated(EnumType.ORDINAL) Color color2;
    @Enumerated(EnumType.STRING) Color color3;
}

enum Color { RED, GREEN, BLUE };
```

In the above example, values of the `color1` and `color2` fields are stored as ordinal numbers (i.e. 0, 1, 2) while values of the `color3` field are stored internally as strings (i.e. “RED”, “GREEN”, “BLUE”).

Serializable Types

Every serializable class (user defined or system defined) is also persistable, but relying on serialization in persisting data has a severe drawback in lack of portability. The internal Java serialization format will be

inaccessible to future versions of ObjectDB on other platform. (e.g. .NET). Therefore, it is recommended to only use explicitly specified persistable types.

Note: Starting ObjectDB version 2.2.9 - serialization is disabled by default.

2.2 JPA Entity Fields

Fields of persistable user defined classes (entity classes, embeddable classes and mapped superclasses) can be classified into the following five groups:

- Transient fields
- Persistent fields
- Inverse (Mapped By) fields
- Primary key (ID) fields
- Version field

The first three groups (transient, persistent and inverse fields) can be used in both entity classes and embeddable classes. However, the last two groups (primary key and version fields) can only be used in entity classes.

Primary key fields are discussed in the Primary Key section.

Transient Fields

Transient entity fields are fields that do not participate in persistence and their values are never stored in the database (similar to transient fields in Java that do not participate in serialization). Static and final entity fields are always considered to be transient. Other fields can be declared explicitly as transient using either the Java `transient` modifier (which also affects serialization) or the JPA `@Transient` annotation (which only affects persistence):

```
@Entity
public class EntityWithTransientFields {
    static int transient1; // not persistent because of static
    final int transient2 = 0;  // not persistent because of final
    transient int transient3; // not persistent because of transient
    @Transient int transient4; // not persistent because of @Transient
}
```

The above entity class contains only transient (non persistent) entity fields with no real content to be stored

in the database.

Persistent Fields

Every non-static non-final entity field is persistent by default unless explicitly specified otherwise (e.g. by using the `@Transient` annotation).

Storing an entity object in the database does not store methods or code. Only the persistent state of the entity object, as reflected by its persistent fields (including persistent fields that are inherited from ancestor classes), is stored.

When an entity object is stored in the database every persistent field must contain either `null` or a value of one of the supported persistable types. ObjectDB supports persistent fields with any declared static type, including a generic `java.lang.Object`, as long as the type of the actual value at runtime is persistable (or `null`).

Every persistent field can be marked with one of the following annotations:

- `OneToOne`, `ManyToOne` - for references of entity types.
- `OneToMany`, `ManyToMany` - for collections and maps of entity types.
- `Basic` - for any other persistable type.

In JPA only `Basic` is optional while the other annotations above are required when applicable. ObjectDB, however, does not enforce using any of these annotations, so they are useful only for classes that are also in use with an ORM JPA provider (such as Hibernate) or to change default field settings. For example:

```
@Entity
public class EntityWithFieldSettings {
    @Basic(optional=false) Integer field1;
    @OneToOne(cascade=CascadeType.ALL) MyEntity field2;
    @OneToMany(fetch=FetchType.EAGER) List<MyEntity> field3;
}
```

The entity class declaration above demonstrates using field and relationship annotations to change the default behavior. `null` values are allowed by default. Specifying `optional=false` (as demonstrated for `field1`) causes an exception to be thrown on any attempt to store an entity with a `null` value in that field. Cascade and fetch settings are explained in chapter 3.

A persistent field whose type is embeddable may optionally be marked with the `@Embedded` annotation, requiring ObjectDB to verify that the type is indeed embeddable:

```
@Entity
public class Person {
    @Embedded Address address;
}
```

Inverse Fields

Inverse (or mapped by) fields contain data that is not stored as part of the entity in the database, but is still available after retrieval by a special automatic query.

Note: Navigation through inverse fields is **much less efficient** than navigation through ordinary persistent fields, since it requires running queries. Inverse fields are essential for collection fields when using ORM JPA implementations, **but not when using ObjectDB**. Avoiding bidirectional relationships and inverse fields, and maintaining two unidirectional relationships is usually much more efficient (unless navigation in the inverse direction is rare).

The following entity classes demonstrate a bidirectional relationship:

```
@Entity
public class Employee {
    String name;
    @ManyToOne Department department;
}

@Entity
public class Department {
    @OneToMany(mappedBy="department") Set<Employee> employees;
}
```

The `mappedBy` element (above) specifies that the `employees` field is an inverse field rather than a persistent field. The content of the `employees` set is not stored as part of a `Department` entity. Instead, employees is automatically populated when a `Department` entity is retrieved from the database. ObjectDB accomplishes this by effectively running the following query (where `:d` represents the `Department` entity):

```
SELECT e FROM Employee e WHERE e.department = :d
```

The `mappedBy` element defines a bidirectional relationship. In a bidirectional relationship, the side that stores the data (the `Employee` class in our example) is the owner. Only changes to the owner side affect the database, since the other side is not stored and calculated by a query.

An index on the owner field may accelerate the inverse query and the load of the inverse field. But even with an index, executing a query for loading a field is relatively slow. Therefore, if the `employees` field is used often, a persistent field rather than inverse field is expected to be more efficient. In this case, two unidirectional and unrelated relationships are managed by the `Employee` and the `Department` classes and the application is responsible to keep them synchronized.

Inverse fields may improve efficiency when managing very large collections that are changed often. This is because a change in the inverse field does not require storing the entire collection again. Only the owner side is stored in the database.

Special settings are available for inverse fields whose type is `List` or `Map`. For an inverse list field, the order of the retrieved owner entities can be set by the `OrderBy` annotation:

```
@Entity
public class Department {
    @OneToMany(mappedBy="department") @OrderBy("name")
    List<Employee> employees;
}
```

In that case the employees field is filled with the results of the following query:

```
SELECT e FROM Employee e WHERE e.department = :d ORDER BY e.name
```

The specified field (`"name"`) must be a sortable field of the owner side.

For an inverse map field, the keys can be extracted from the inverse query results by specifying a selected key field using the MapKey annotation:

```
@Entity
public class Department {
    @OneToMany(mappedBy="department") @MapKey(name="name")
    Map<String,Employee> employees;
}
```

The employees map is filled with a mapping of employee names to the `Employee` objects to which they pertain.

Single value inverse fields are also supported:

```
@Entity
public class Employee {
    @OneToOne MedicalInsurance medicalInsurance;
}

@Entity
public class MedicalInsurance {
    @OneToOne(mappedBy="medicalInsurance") Employee employee;
}
```

A single value inverse field is less efficient than an inverse collection or map field because no proxy class is used and the inverse query is executed eagerly when the entity object is first accessed.

Version Field

ObjectDB maintains a version number for every entity object. The initial version of a new entity object (when stored in the database for the first time) is 1. For every transaction in which an entity object is modified its version number is automatically increased by one. Version fields are used in conjunction with optimistic locking (as explained in the Locking in JPA section in chapter 3).

You can expose entity object versions and make their values accessible to your application by marking the version fields in your entity classes with the `Version` annotation. In addition, a version field should have a numeric type:

```
@Entity public class EntityWithVersionField {
     @Version long version;
}
```

If a version field exists, ObjectDB automatically injects the version value into that field. Version fields should be treated as read only by the application and no mutator methods should be written against them. Only one version field per entity class is allowed. In fact, a single version field per entity class hierarchy is sufficient because a version field is inherited by subclasses.

Unlike ORM JPA providers, ObjectDB always manages versions for entity objects, regardless as to whether or not a version field is explicitly defined. Therefore, optimistic locking is supported by ObjectDB even when a version field is not defined. Nevertheless, defining a version field has some advantages:

- The application becomes more portable (to ORM-based JPA implementations).
- Even when entity object versions are not in use directly by the application, exposing their values might be useful occasionally for debugging and logging.

- Version values cannot be preserved for detached entity objects (explained in chapter 3) unless either the entity class is enhanced (explained in chapter 5) or a version field is explicitly defined.

Property Access

When an entity is being stored to the database data is extracted from the persistent fields of that entity. Likewise, when an entity is retrieved the persistent fields are initialized with data from the database.

By default, ObjectDB accesses the fields directly, but accessing fields indirectly as properties using get and set methods is also supported. To use property access mode, every non-transient field must have get and set methods based on the Java bean property convention.

Property access is enabled by moving all the JPA annotations from the fields to their respective get methods and specifying the Access annotation on the class itself:

```
@Entity @Access(AccessType.PROPERTY)
public static class PropertyAccess {
    private int _id;
    @Id int getId() { return _id; }
    void setId(int id) { _id = id; }

    private String str;
    String getStr() { return str; }
    void setStr(String str) { this.str = str; }
}
```

In some JPA implementations, such as Hibernate, using property access may have some performance benefits. This is not the case with ObjectDB. Therefore, considering the extra complexity that is involved in setting up and maintaining property access, the default field access mode is usually preferred.

2.3 JPA Primary Key

Every entity object that is stored in the database has a primary key. Once assigned, the primary key cannot be modified. It represents the entity object as long as it exists in the database.

As an object database, ObjectDB supports implicit object IDs, so an explicitly defined primary key is not required. But ObjectDB also supports explicit standard JPA primary keys, including sequential value generation. This is a very powerful feature of ObjectDB that is absent from other object oriented databases.

Entity Identification

Every entity object in the database is uniquely identified (and can be retrieved from the database) by the combination of its type and its primary key. Primary key values are unique per entity class. Instances of different entity classes, however, may share the same primary key value.

Only entity objects have primary keys. Instances of other persistable types are always stored as part of their containing entity objects and do not have their own separate identity.

Automatic Primary Key

By default the primary key is a sequential 64 bit number (`long`) that is set automatically by ObjectDB for every new entity object that is stored in the database. The primary key of the first entity object in the database is 1, the primary key of the second entity object is 2, etc. Primary key values are **not** recycled when entity objects are deleted from the database.

The primary key value of an entity can be accessed by declaring a primary key field:

```
@Entity
public class Project {
    @Id @GeneratedValue long id; // still set automatically
     :
}
```

The `@Id` annotation marks a field as a primary key field. When a primary key field is defined the primary key value is automatically injected into that field by ObjectDB.

The `@GeneratedValue` annotation specifies that the primary key is automatically allocated by ObjectDB. Automatic value generation is discussed in detail in the Generated Values section.

Application Set Primary Key

If an entity has a primary key field that is not marked with `@GeneratedValue`, automatic primary key value is not generated and the application is responsible to set a primary key by initializing the primary key field. That must be done before any attempt to persist the entity object:

```
@Entity
public class Project {
    @Id long id; // must be initialized by the application
     :
```

```
}
```

A primary key field that is set by the application can have one of the following types:

- Primitive types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`.
- Equivalent wrapper classes from package java.lang:
  `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.
- `java.math.BigInteger`, `java.math.BigDecimal`.
- `java.lang.String`.
- `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`.
- Any enum type.
- Reference to an entity object.

Composite Primary Key

A composite primary key consist of multiple primary key fields. Each primary key field must be one of the supported types listed above.

For example, the primary key of the following `Project` entity class consists of two fields:

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id int departmentId;
    @Id long projectId;
     :
}
```

When an entity has multiple primary key fields, JPA requires defining a special ID class that is attached to the entity class using the `@IdClass` annotation. The ID class reflects the primary key fields and its objects can represent primary key values:

```
Class ProjectId {
    int departmentId;
    long projectId;
}
```

ObjectDB does not enforce defining ID classes. However, an ID class is required if entity objects have to be retrieved by their primary key as shown in the Retrieving Entities section.

Embedded Primary Key

An alternate way to represent a composite primary key is to use an embeddable class:

```
@Entity
public class Project {
    @EmbeddedId ProjectId id;
     :
}

@Embeddable
Class ProjectId {
    int departmentId;
    long projectId;
}
```

The primary key fields are defined in an embeddable class. The entity contains a single primary key field that is annotated with `@EmbeddedId` and contains an instance of that embeddable class. When using this form a separate ID class is not defined because the embeddable class itself can represent complete primary key values.

Obtaining the Primary Key

JPA 2 provides a generic method for getting the object ID (primary key) of a specified managed entity object. For example:

```
PersistenceUnitUtil util = emf.getPersistenceUnitUtil();
Object projectId = util.getIdentifier(project);
```

A `PersistenceUnitUtil` instance is obtained from the `EntityManagerFactory`. The getIdentifier method takes one argument, a managed entity object, and returns the primary key. In case of a composite primary key - an instance of the ID class or the embeddable class is returned.

Using Primary Keys for Object Clustering

Entity objects are physically stored in the database ordered by their primary key. Sometimes it is useful to choose a primary key that helps clustering entity objects in the database in an efficient way. This is especially useful when using queries that return large result sets.

As an example, consider a real time system that detects events from various sensors and stores the details

in a database. Each event is represented by an `Event` entity object that holds time, sensor ID and additional details. Suppose that queries that retrieve all the events of a specified sensor in a specified period are common and return thousands of `Event` objects. In that case the following primary key can significantly improve query run performance:

```
@Entity
public class Event {
    @EmbeddedId EventId id;
     :
}

@Embeddable
Class EventId {
    int sensorId;
    Date time;
}
```

Because entity objects are ordered in the database by their primary key, events of the same sensor during a period of time are stored continuously and can be collected by accessing a minimum number of database pages.

On the other end, such a primary key requires more storage space (especially if there are many references to `Event` objects in the database because references to entities hold primary key values) and is less efficient in store operations. Therefore, all factors have to be considered and a benchmark might be needed to evaluate the different alternatives in order to select the best solution.

2.4 Auto Generated Values

Marking a field with the `@GeneratedValue` annotation specifies that a value will be automatically generated for that field. This is primarily intended for primary key fields but ObjectDB also supports this annotation for non-key numeric persistent fields as well. Several different value generation strategies can be used as explained below.

The Auto Strategy

ObjectDB maintains a special global number generator for every database. This number generator is used to generate automatic object IDs for entity objects with no primary key fields defined (as explained in the previous section).

The same number generator is also used to generate numeric values for primary key fields annotated by `@GeneratedValue` with the `AUTO` strategy:

```
@Entity
public class EntityWithAutoId1 {
    @Id @GeneratedValue(strategy=GenerationType.AUTO) long id;
     :
}
```

`AUTO` is the default strategy, so the following definition is equivalent:

```
@Entity
public class EntityWithAutoId2 {
    @Id @GeneratedValue long id;
     :
}
```

During a commit the `AUTO` strategy uses the global number generator to generate a primary key for every new entity object. These generated values are unique at the database level and are never recycled, as explained in the previous section.

The Identity Strategy

The `IDENTITY` strategy is very similar to the `AUTO` strategy:

```
@Entity
public class EntityWithIdentityId {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY) long id;
     :
}
```

The `IDENTITY` strategy also generates an automatic value during commit for every new entity object. The difference is that a separate identity generator is managed per type hierarchy, so generated values are unique only per type hierarchy.

The Sequence Strategy

The sequence strategy consists of two parts - defining a named sequence and using the named sequence in one or more fields in one or more classes. The `@SequenceGenerator` annotation is used to define a

sequence and accepts a name, an initial value (the default is 1) and an allocation size (the default is 50). A sequence is global to the application and can be used by one or more fields in one or more classes. The SEQUENCE strategy is used in the @GeneratedValue annotation to attach the given field to the previously defined named sequence:

```
@Entity
// Define a sequence - might also be in another class:
@SequenceGenerator(name="seq", initialValue=1, allocationSize=100)
public class EntityWithSequenceId {
    // Use the sequence that is defined above:
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")
    @Id long id;
}
```

Unlike AUTO and IDENTITY, the SEQUENCE strategy generates an automatic value as soon as a new entity object is persisted (i.e. before commit). This may be useful when the primary key value is needed earlier. To minimize round trips to the database server, IDs are allocated in groups. The number of IDs in each allocation is specified by the allocationSize attribute. It is possible that some of the IDs in a given allocation will not be used. Therefore, this strategy does not guarantee there will be no gaps in sequence values.

The Table Strategy

The TABLE strategy is very similar to the SEQUENCE strategy:

```
@Entity
@TableGenerator(name="tab", initialValue=0, allocationSize=50)
public class EntityWithTableId {
    @GeneratedValue(strategy=GenerationType.TABLE, generator="tab")
    @Id long id;
}
```

ORM-based JPA providers (such as Hibernate, TopLink, EclipseLink, OpenJPA, JPOX, etc.) simulate a sequence using a table to support this strategy. ObjectDB does not have tables, so the TABLE and SEQUENCE strategies are almost identical.

A tiny difference is related to the initial value attribute. Whereas the SEQUENCE strategy maintains the next sequence number to be used the TABLE strategy maintains the last value that was used. The implication for the initialValue attribute is that if you want sequence numbers to start with 1 in the TABLE strategy

`initialValue=0` has to be specified in the `@SequenceGenerator` annotation.

2.5 Index Definition

Querying without indexes requires iteration over entity objects in the database one by one.
This may take a significant amount of time if many entity objects have to be examined. Using proper indexes the iteration can be avoided and complex queries over millions of objects can be executed quickly. Index management introduces overhead in terms of maintenance time and storage space, so deciding which fields to define with indexes should be done carefully.

Single Field Index

JPA does not define a standard method for declaring indexes, but JDO does. The following entity definition uses JDO's `@Index` and `@Unique` annotations to define indexes:

```
import javax.jdo.annotations.Index;
import javax.jdo.annotations.Unique;

@Entity
public class EntityWithSimpleIndex {
    @Index String indexedField1;
    @Index(unique="true") int indexedField2; // unique
    @Index(name="i3") int indexedField3;
    @Unique Integer indexedField4; // unique
    @Unique(name="u2") Date indexedField5; // unique
}
```

`@Unique` represents a unique index that prevents duplicate values in the indexed field.
A `PersistenceException` is thrown on commit (or flush) if different entities have the same value in a unique field (similar to how primary keys behave).

`@Index` represents either an ordinary index with no unique constraint or a unique index if `unique="true"` is specified (the default is false).

The optional `name` attribute has no specific role but might be presented in the ObjectDB Explorer and in logging.

When an entity object is stored in the database every indexed field must contain either `null` or a value of one of the following persistable types:

- Primitive types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`.
- Equivalent wrapper classes from package java.lang:
  `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.
- `java.math.BigInteger`, `java.math.BigDecimal`.
- `java.lang.String`.
- `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`.
- Any enum type.
- Reference to an entity object.
- Arrays and collections that contain values of the above types (including `null`).

Indexes can only be defined on ordinary persistent fields (not on primary key / version fields).

Composite Index

A composite index is an index on more than one persistent field. It is defined by specifying multiple fields in the `members` attribute of the the `@Index` or `@Unique` annotations:

```
@Entity
@Index(members={"lastName","firstName"})
public class EntityWithCompositeIndex {
    String firstName;
    String lastName;
}
```

When indexed fields are specified explicitly in the `members` attribute, as shown above, the `@Index` or `@Unique` annotation can be attached to either the class or to any persistent field in the class.

Multiple `@Index` annotations can be wrapped with an `@Indices` annotation:

```
@Entity
@Indices({
    @Index(members={"lastName","firstName"})
    @Index(members={"firstName"}, unique="true")
}
public class EntityWithCompositeIndex {
    String firstName;
    String lastName;
}
```

Similarly, the `@Uniques` annotation can wrap multiple `@Unique` annotations.

As shown above, the `members` attribute can also be used for a single field index. This is useful if you want centralize your index definitions at the top of the class and is equivalent to omitting `members` and attaching the `@Index` annotation directly to the indexed field.

Multi Part Path Index

The `members` attribute is also required in order to define indexes on multi part paths:

```
@Entity
@Index(members={"address.city"})
public class Employee {
    String firstName;
    String lastName;
    Address address;
     :
}

@Embeddable
class Address {
    String street;
    String city;
     :
}
```

Indexes must always refer to values that are stored as part of the entity. Therefore, indexes on multi part paths are only allowed when using embeddable classes as fields of embedded object are stored as part of the containing entity.

Composite indexes on multi part paths are also allowed:

```
@Entity
@Index(members={"addresses.city,addresses.street"})
public class Employee {
     :
    List<Address> addresses;
     :
}
```

Notice that the paths include a collection, so multiple values will be maintained by the index for every entity.

Multi part paths in a composite index must have the same length. Therefore, the following index definition is

invalid:

```
@Entity
@Index(members={"lastName", "address.city"}) // INVALID
public class Employee {
    String firstName;
    String lastName;
    Address address;
     :
}
```

Indexes in Queries

ObjectDB manages a BTree for every index. A BTree is an ordered map data structure that ObjectDB maintains in the file system rather than in memory. The keys of the BTree are all the unique values in the indexed field (or arrays of values in composite indexes), in all the entities of that class (including subclasses). Every key is associated with a list of references to the entities that contain that value.

Indexes require maintenance time and consume storage space. Therefore, using indexes wisely requires an understanding of how ObjectDB uses indexes to accelerate query execution.

Indexes are especially efficient in lookup and range queries:

```
SELECT p FROM Point p WHERE p.x = 100
SELECT p FROM Point p WHERE p.x BETWEEN 50 AND 80
SELECT p FROM Point p WHERE p.x >= 50 AND p.x <= 80
```

By using an index on field x, ObjectDB can find the results using a range scan, which is very efficient because only branches of the BTree that are relevant are iterated.

A composite index on fields x and y can also be used for the queries above. In this case the order of the fields in the composite index definition is important, because it determines which field is used as the BTree's primary sort key and which field is used as a secondary sort key. If x is specified first, the BTree is ordered by x values, so a range scan on x values is supported. On the other hand, if y is specified first, the BTree is ordered by y values, so a full BTree scan is required to find all the objects with relevant x values. A full index scan is less efficient than a range scan, but might be still more efficient than iteration over the entities data.

A composite index on fields x and y enables quick execution of the following queries:

```
SELECT p FROM Point p WHERE p.x = 100 AND p.y = 100
```

```
SELECT p FROM Point p WHERE p.x = 100 AND p.y BETWEEN 50 AND 80
```

For the second query above, the index order should be x first and y second.

Indexes on collections are useful in JOIN queries:

```
SELECT d FROM Document d JOIN d.words w WHERE w = 'JPA'
```

The `Document` entity class contains a `words` field whose type is `List<String>`. The query retrieves the documents that contain "JPA". An index on `words` will manage for every word all the documents that contain it. Therefore, using such index, the query above can be executed by a quick index range scan.

The same collection index can also be used for executing the following query:

```
SELECT d FROM Document d JOIN d.words w WHERE LENGTH(w) >= 10
```

But this time a full index scan is required because the index uses the lexicographic order of the words and is not ordered by the length of the words.

The bottom line is that if there is an index that contains all the fields in the WHERE clause of a given query that query will run faster as it will be able to, at the very least, utilize a full index scan. Even better, if the field order as defined in the index matches the field order used in the WHERE clause of the query a more efficient range scan can be performed.

ObjectDB also uses indexes for sorting results and for projection:

```
SELECT MIN(p.x) FROM Point p WHERE p.x < p.y ORDER BY p.y
```

In the above example, a composite index on fields x and y would cover all the fields in the query (including the WHERE and ORDER BY clauses) saving the need to access the entities themselves. Instead, because of the comparison in the WHERE clause, a full index scan would be performed. Additionally, if field y was the first field defined in the index the results would already be produced in the requested order (the order of the scan) thus eliminating the need for a separate sort.

Finally, indexes are also used in MIN and MAX queries:

```
SELECT MIN(p.x), MAX(p.x) FROM Point p
```

Given an index on field x ObjectDB can simply return the first and last key in the BTree, without any iteration.

2.6 Database Schema Evolution

Modifications to entity classes that do not change their persistent field definitions (their schema) are transparent to ObjectDB. This includes adding, removing and modifying constructors, methods and non persistent fields. However, additions, deletions and modifications to the persistent fields of an entity class are detected by ObjectDB. New entity objects have to be stored in the new class schema, and old entity objects, which were stored previously in the old class schema, have to be converted to the new schema.

Note: In client-server mode the ObjectDB server must be restarted after a schema change.

Automatic Schema Evolution

ObjectDB implements an automatic schema evolution mechanism that enables transparent use of old entity objects after schema change. When an entity object of an old schema is loaded into memory it is automatically converted into an instance of the up to date entity class. This is done automatically in memory each time the entity object is loaded. The database object is only updated to the new schema when that entity object is stored to the database again.

Conversion of an entity object to the new schema is done on a field by field basis:

- For every field in the new schema for which there is a *matching field* in the old schema, the new field in the new entity object is initialized using the value of the matching old field in the original entity object.
- Fields in the new schema that do not have matching fields in the old schema are initialized with default values (`0`, `false` or `null`).
- Fields in the old schema that do not have matching fields in the new schema are simply ignored (and their content is lost).

A matching field is a field with the same name and either the same type or a convertible type, as explained below. A matching field might also be located in a different place in the class hierarchy. That makes automatic schema evolution very flexible and almost insensitive to class hierarchy changes (e.g. moving fields between classes in the hierarchy, removing an intermediate class in the hierarchy, etc.).

Convertible Types

When an old matching field is found but its type is different than the type of the new field (with the same name), a conversion is required. If the old type is inconvertible to the new type (for instance a change from `int` to `Date`) the fields are not considered as matching and the new field is initialized with a default value (`0`,

`false` or `null`).

The following type conversions are supported:

- From any numeric type to any numeric type. In this context numeric types are: `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`, `BigInteger`, `BigDecimal` and enum values that are stored as numeric ordinal values (the default).
- From any type to Boolean or boolean (`0`, `null` and `false` are converted to `false`, any other value is converted to `true`).
- From any type to `String` (using `toString()` if necessary).
- From `String` to numeric types including enum types (when applicable).
- From any date type to any date type.
- From any collection or array type to any collection or array type, as long as the elements are convertible (e.g. from `int[]` to `ArrayList<Long>`).
- From any object to any collection or array that can contain that object as en element.
- From any map type to any map type as long as the keys and values are convertible (e.g. from `HashMap<Long,Long>` to `TreeMap`).
- Any other conversion that is a valid casting operation in Java.

Renaming (Package, Class and Field)

The automatic schema evolution mechanism, as described above, is based on matching fields by their names. When schema upgrade includes also renaming fields, classes or packages, these changes must be specified explicitly in the configuration to avoid data loss. The Schema Update section in chapter 6 explains how to specify such changes in the configuration file.

2.7 JPA Persistence Unit

A JPA Persistence Unit is a logical grouping of user defined persistable classes (entity classes, embeddable classes and mapped superclasses) with related settings. Defining a persistence unit is optional when using ObjectDB, but required by JPA.

persistence.xml

Persistence units are defined in a `persistence.xml` file, which has to be located in the `META-INF` directory in the classpath. One `persistence.xml` file can include definitions for one or more persistence units. The portable way to instantiate an `EntityManagerFactory` in JPA (as explained in the JPA Overview

section) requires a persistence unit.

The following `persistence.xml` file defines one persistence unit:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

   <persistence-unit name="my-pu">
     <description>My Persistence Unit</description>
     <provider>com.objectdb.jpa.Provider</provider>
     <mapping-file>META-INF/mappingFile.xml</mapping-file>
     <jar-file>packedEntity.jar</jar-file>
     <class>sample.MyEntity1</class>
     <class>sample.MyEntity2</class>
     <properties>
       <property name="javax.persistence.jdbc.url"
                 value="objectdb://localhost/my.odb"/>
       <property name="javax.persistence.jdbc.user" value="admin"/>
       <property name="javax.persistence.jdbc.password" value="admin"/>
     </properties>
   </persistence-unit>

</persistence>
```

A persistence unit is defined by a `persistence-unit` XML element. The required `name` attribute (“`my-pu`” in the example) identifies the persistence unit when instantiating an `EntityManagerFactory`. It may also have optional sub elements:

- The provider element indicates which JPA implementation should be used. ObjectDB is represented by the `com.objectdb.jpa.Provider` string. If not specified, the first JPA implementation that is found is used.
- The mapping-file elements specify XML mapping files that are added to the default `META-INF/orm.xml` mapping file. Every annotation that is described in this manual can be replaced by equivalent XML in the mapping files (as explained below).
- The `jar-file` elements specify JAR files that should be searched for managed persistable classes.
- The `class` elements specify names of managed persistable classes (see below).
- The `property` elements specify general properties. JPA 2 defines standard properties for specifying database url, username and password, as demonstrated above.

XML Mapping Metadata

ObjectDB supports using XML metadata as an alternative to annotations. Both JPA mapping files and JDO `package.jdo` files are supported. This manual focuses on using annotations which are more common and usually more convenient. Details on using XML metadata can be found in the JPA and JDO specifications and books.

Managed Persistable Classes

JPA requires registration of all the user defined persistable classes (entity classes, embeddable classes and mapped superclasses), which are referred to by JPA as managed classes, as part of a persistence unit definition.

Classes that are mentioned in mapping files as well as annotated classes in the JAR that contains the `persistence.xml` file (if it is packed) are registered automatically. If the application is not packed in a JAR yet, ObjectDB (as an extension) automatically registers classes under the classpath root directory that contains the `META-INF/persistence.xml` file. Other classes have to be registered explicitly by using `class` elements (for single class registration) or `jar-file` elements (for registration of all the classes in the jar file).

ObjectDB does not enforce registration of all the managed classes. However, it might be useful to register classes that define generators and named queries (by annotations). Otherwise, the generators and named queries are available only when the containing classes become known to ObjectDB, for example when a first instance of the class is stored in the database.

Chapter 3 - Using JPA

This chapter explains how to manage ObjectDB databases using the Java Persistence API (JPA).

The first two pages introduce basic JPA interfaces and concepts:

- Database Connection using JPA
- Working with JPA Entity Objects

The next section explains how to use JPA for database CRUD operations:

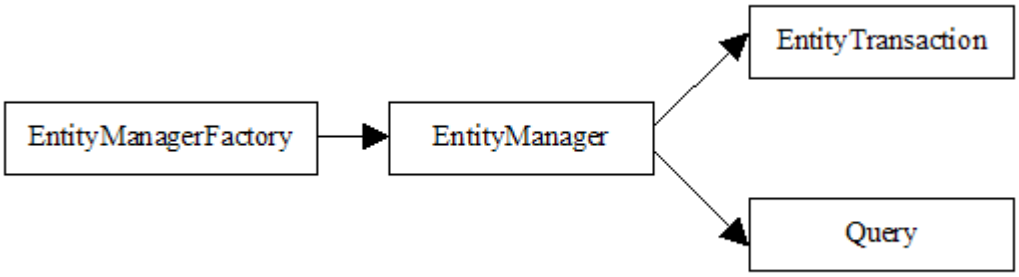
- CRUD Operations with JPA

More advanced topics (e.g. locking and events) are discussed in the last section:

- Advanced JPA Topics

3.1 Database Connection using JPA

Working with the Java Persistence API (JPA) consists of using the following interfaces:



Overview

A connection to a database is represented by an `EntityManager` instance, which also provides functionality for performing operations on a database. Many applications require multiple database connections during their lifetime. For instance, in a web application it is common to establish a separate database connection, using a separate `EntityManager` instance, for every HTTP request.

The main role of an `EntityManagerFactory` instance is to support instantiation of `EntityManager` instances. An `EntityManagerFactory` is constructed for a specific database, and by managing resources efficiently (e.g. a pool of sockets), provides an efficient way to construct multiple `EntityManager` instances for that database. The instantiation of the `EntityManagerFactory` itself might be less efficient, but it is a one time operation. Once constructed, it can serve the entire application.

Operations that modify the content of a database require active transactions. Transactions are managed by an `EntityTransaction` instance obtained from the `EntityManager`.

An `EntityManager` instance also functions as a factory for `Query` instances, which are needed for executing queries on the database.

Every JPA implementation defines classes that implement these interfaces. When you use ObjectDB you work with instances of ObjectDB classes that implement these interfaces, and because standard JPA interfaces are used your application is portable.

EntityManagerFactory

An `EntityManagerFactory` instance is obtained by using a static factory method of the JPA bootstrap class, `Persistence`:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("myDbFile.odb");
```

Another form of the `createEntityManagerFactory` method takes a map of persistence unit properties as a second parameter:

```
Map<String, String> properties = new HashMap<String, String>();
properties.put("javax.persistence.jdbc.user", "admin");
properties.put("javax.persistence.jdbc.password", "admin");
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost:6136/myDbFile.odb", properties);
```

The `EntityManagerFactory` instance, when constructed, opens the database. If the database does not yet exist a new database file is created.

When the application is finished using the `EntityManagerFactory` it has to be closed:

```
emf.close();
```

Closing the `EntityManagerFactory` closes the database file.

Connection URL

The `createEntityManagerFactory` method takes as an argument a name of a persistence unit.

As an extension, ObjectDB enables specifying a database url (or path) directly, bypassing the need for a persistence unit. Any string that starts with `objectdb:` or ends with `.odb` or `.objectdb` is considered by ObjectDB to be a database url rather than as a persistence unit name.

To use ObjectDB embedded directly in your application (embedded mode), an absolute path or a relative path of a local database file has to be specified (e.g. `"my.odb"`). Specifying the `objectdb:` protocol as a prefix (e.g. `"objectdb:my.odb"`) is optional if the database file name extension is `odb` or `objectdb` and required for other file name extensions (e.g. `"objectdb:my-db.tmp"`).

To use client server mode, a url in the format `objectdb://host:port/path` has to be specified.
In this case, an ObjectDB Database Server is expected to be running on a machine named host (could be domain name or IP address) and listening on the specified port (the default is 6136 when not specified). The path indicates the location of the database file on the server, relative to the server data root path.

Connection URL Parameters

The following parameters are supported as part of an ObjectDB connection url:

- `user` - for specifying a username in client server mode.
- `password` - for specifying a user password in client server mode.
- `drop` - for deleting any existing database content (useful for tests).

To connect to an ObjectDB server registered username and password have to be specified:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost/myDbFile.odb;user=admin;password=admin");
```

This is equivalent to specifying a username and a password in the persistence unit or in a map of properties (as demonstrated above).

To obtain a connection to an empty database (discarding existing content if any) the `drop` parameter has to be specified:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("objectdb:myDbFile.tmp;drop");
```

Getting an empty clean database easily is very useful in tests. However, to avoid the risk of losing data - the `drop` parameter is ignored unless the database file name extension indicates that it is a temporary database. By default, the `tmp` and `temp` file name extensions represent temporary databases that can be dropped, but this can be configured.

EntityManager

An `EntityManager` instance may represent either a remote connection to a remote database server (in client-server mode) or a local connection to a local database file (in embedded mode). The functionality in both cases is the same. Given an `EntityManagerFactory emf`, a short term connection to the database might have the following form:

```
EntityManager em = emf.createEntityManager();
try {
    // TODO: Use the EntityManager to access the database
}
finally {
    em.close();
}
```

The `EntityManager` instance is obtained from the owning `EntityManagerFactory` instance. Calling the `close` method is essential to release resources (such as a socket in client-server mode) back to the owning `EntityManagerFactory`.

`EntityManagerFactory` defines another method for instantiation of `EntityManager` that, like the factory, takes a map of properties as an argument. This form is useful when a user name and a password other than the `EntityManagerFactory`'s default user name and password have to specified:

```
Map<String, String> properties = new HashMap<String, String>();
properties.put("javax.persistence.jdbc.user", "user1");
properties.put("javax.persistence.jdbc.password", "user1pwd");
EntityManager em = emf.createEntityManager(properties);
```

EntityTransaction

Operations that affect the content of the database (store, update, delete) must be performed within an active transaction. The `EntityTransaction` interface represents and manages database transactions. Every `EntityManager` holds a single attached `EntityTransaction` instance that is available via the `getTransaction` method:

```
try {
    em.getTransaction().begin();
    // Operations that modify the database should come here.
    em.getTransaction().commit();
}
```

```
finally {
    if (em.getTransaction().isActive())
        em.getTransaction().rollback();
}
```

A transaction is started by a call to `begin` and ended by a call to either `commit` or `rollback`. All the operations on the database within these boundaries are associated with that transaction and are kept in memory until the transaction is ended. If the transaction is ended with a `rollback`, all the modifications to the database are discarded. However, by default, the in-memory instance of the managed entity is not affected by the rollback and is not returned to its pre-modified state.

Ending a transaction with a `commit` propagates all the modifications physically to the database. If for any reason a `commit` fails, the transaction is rolled back automatically (including rolling back modifications that have already been propagated to the database prior to the failure) and a `RollbackException` is thrown.

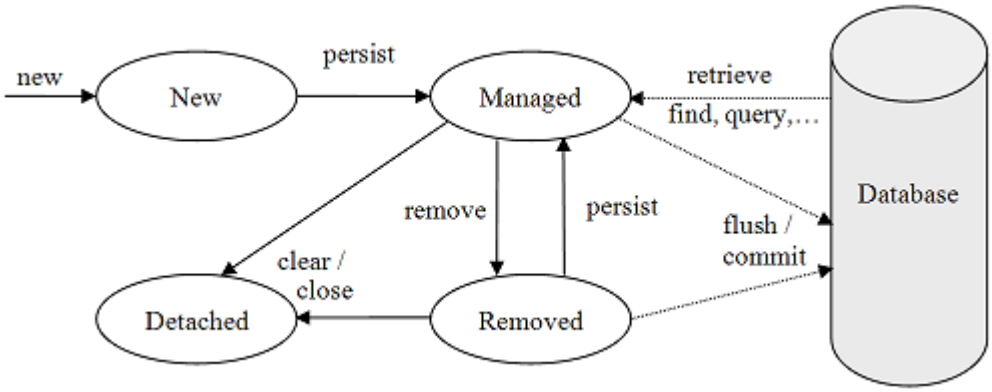
3.2 Working with JPA Entity Objects

Entity objects are in-memory instances of entity classes (persistable user defined classes), which can represent physical objects in the database.

Managing an ObjectDB Object Database using JPA requires using entity objects for many operations, including storing, retrieving, updating and deleting database objects.

Entity Object Life Cycle

The life cycle of entity objects consists of four states: New, Managed, Removed and Detached.



When an entity object is initially created its state is **New**. In this state the object is not yet associated with an `EntityManager` and has no representation in the database.

An entity object becomes **Managed** when it is persisted to the database via an `EntityManager's persist` method, which must be invoked within an active transaction. On transaction commit, the owning `EntityManager` stores the new entity object to the database. More details on storing objects are provided in the Storing Entities section.

Entity objects retrieved from the database by an `EntityManager` are also in the **Managed** state. Object retrieval is discussed in more detail in the Retrieving Entities section.

If a managed entity object is modified within an active transaction the change is detected by the owning `EntityManager` and the update is propagated to the database on transaction commit. See the Updating Entities section for more information about making changes to entities.

A managed entity object can also be retrieved from the database and marked for deletion, by using the `EntityManager's remove` method within an active transaction. The entity object changes its state from Managed to **Removed**, and is physically deleted from the database during commit. More details on object deletion are provided in the Deleting Entities section.

The last state, **Detached**, represents entity objects that have been disconnected from the `EntityManager`. For instance, all the managed objects of an `EntityManager` become detached when the `EntityManager` is closed. Working with detached objects, including merging them back to an `EntityManager`, is discussed in the Detached Entities section.

The Persistence Context

The persistence context is the collection of all the managed objects of an `EntityManager`. If an entity object that has to be retrieved already exists in the persistence context, the existing managed entity object is returned without actually accessing the database (except retrieval by `refresh`, which always requires accessing the database).

The main role of the persistence context is to make sure that a database entity object is represented by no more than one in-memory entity object within the same `EntityManager`. Every `EntityManager` manages its own persistence context. Therefore, a database object can be represented by different memory entity objects in different `EntityManager` instances. But retrieving the same database object more than once using the same `EntityManager` should always result in the same in-memory entity object.

Another way of looking at it is that the persistence context also functions as a local cache for a given `EntityManager`. ObjectDB also manages a level 2 shared cache for the `EntityManagerFactory` as well as other caches as explained in the Configuration chapter.

By default, managed entity objects that have not been modified or removed during a transaction are held in the persistence context by weak references. Therefore, when a managed entity object is no longer in use by

the application the garbage collector can discard it and it is automatically removed from the persistence context. ObjectDB can be configured to use strong references or soft references instead of weak references.

The `contains` method can check if a specified entity object is in the persistence context:

```
boolean isManaged = em.contains(employee);
```

The persistence context can be cleared by using the `clear` method, as so:

```
em.clear();
```

When the persistence context is cleared all of its managed entities become detached and any changes to entity objects that have not been flushed to the database are discarded. Detached entity objects are discussed in more detail in the Detached Entities section.

3.3 CRUD Operations with JPA

The following subsections explain how to use JPA for CRUD database operations:

- Storing JPA Entity Objects
- Retrieving JPA Entity Objects
- Updating JPA Entity Objects
- Deleting JPA Entity Objects

3.3.1 Storing JPA Entity Objects

New entity objects can be stored in the database either explicitly by invoking the `persist` method or implicitly as a result of a cascade operation.

Explicit Persist

The following code stores an instance of the `Employee` entity class in the database:

```
Employee employee = new Employee("Samuel", "Joseph", "Wurzelbacher");
em.getTransaction().begin();
em.persist(employee);
```

```
em.getTransaction().commit();
```

The Employee instance is constructed as an ordinary Java object and its initial state is New. An explicit call to persist associates the object with an owner EntityManager em and changes its state to Managed. The new entity object is stored in the database when the transaction is committed.

An IllegalArgumentException is thrown by persist if the argument is not an instance of an entity class. Only instances of entity classes can be stored in the database independently. Objects of other persistable types can only be stored in the database embedded in containing entities (as field values).

A TransactionRequiredException is thrown if there is no active transaction when persist is called because operations that modify the database require an active transaction.

If the database already contains another entity of the same type with the same primary key, an EntityExistsException is thrown. The exception is thrown either by persist (if that existing entity object is currently managed by the EntityManager) or by commit.

Referenced Embedded Objects

The following code stores an Employee instance with a reference to an Address instance:

```
Employee employee = new Employee("Samuel", "Joseph", "Wurzelbacher");
Address address = new Address("Holland", "Ohio");
employee.setAddress(address);

em.getTransaction().begin();
em.persist(employee);
em.getTransaction().commit();
```

Instances of persistable types other than entity classes are automatically stored embedded in containing entity objects. Therefore, if Address is defined as an embeddable class the Employee entity object is automatically stored in the database with its Address instance as an embedded object.

Notice that embedded objects cannot be shared by multiple entity objects. Each containing entity object should have its own embedded objects.

Referenced Entity Objects

On the other hand, suppose that the Address class in the code above is defined as an entity class. In this case, the referenced Address instance is not stored in the database automatically with the referencing

Employee instance.

To avoid a dangling reference in the database, an IllegalStateException is thrown on commit if a persisted entity object has to be stored in the database in a transaction and it references another entity object that is not expected to be stored in the database at the end of that transaction.

It is the application's responsibility to verify that when an object is stored in the database, the entire closure of entity objects that are reachable from that object by navigation through persistent reference fields is also stored in the database. This can be done either by explicit persist of every reachable object or alternatively by setting automatic cascading persist.

Cascading Persist

Marking a reference field with CascadeType.PERSIST (or CascadeType.ALL that also covers PERSIST) indicates that persist operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.<code>PERSIST</code>)
    private Address address;
     :
}
```

In the example above, the Employee entity class contains an address field that references an instance of Address, which is another entity class. Due to the CascadeType.PERSIST setting, when an Employee instance is persisted the operation is automatically cascaded to the referenced Address instance which is then automatically persisted without the need for a separate persist call for Address. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, etc.).

Global Cascading Persist

Instead of specifying CascadeType.PERSIST individually for every relevant reference field, it can be specified globally for any persistent reference, either by setting the ObjectDB configuration or in a JPA portable way, by specifying the cascade-persist XML element in the XML mapping file:

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
```

```
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
   <persistence-unit-metadata>
     <persistence-unit-defaults>
       <cascade-persist/>
     </persistence-unit-defaults>
   </persistence-unit-metadata>
</entity-mappings>
```

The mapping file has to be located either in the default location, `META-INF/orm.xml`, or in another location that is specified explicitly in the persistence unit definition (in `persistence.xml`).

Batch Store

Storing a large number of entity objects requires special consideration. The combination of the `clear` and `flush` methods can be used to save memory in large transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.flush();
        em.clear();
    }
}
em.getTransaction().commit();
```

Managed entity objects consume more memory than ordinary non managed Java objects. Therefore, holding 1,000,000 managed `Point` instances in the persistence context might consume too much memory. The sample code above clears the persistence context after every 10,000 persists. Updates are flushed to the database before clearing, otherwise they would be lost.

Updates that are sent to the database using `flush` are considered temporary and are only visible to the owner `EntityManager` until a commit. With no explicit `commit`, these updates are later discarded. The combination of `clear` and `flush` enables moving the temporary updates from memory to the database.

Note: Flushing updates to the database is sometimes also useful before executing queries in order to get up to date results.

Storing large amount of entity objects can also be performed by multiple transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.getTransaction().commit();
        em.clear();
        em.getTransaction().begin();
    }
}
em.getTransaction().commit();
```

Splitting a batch store into multiple transactions is more efficient than using one transaction with multiple invocations of the `flush` and `clear` methods. So using multiple transactions is preferred when applicable.

3.3.2 Retrieving JPA Entity Objects

The Java Persistence API (JPA) provides various ways to retrieve objects from the database. The retrieval of objects does not require an active transaction because it does not change the content of the database.

The persistence context serves as a cache of retrieved entity objects. If a requested entity object is not found in the persistence context a new object is constructed and filled with data that is retrieved from the database (or from the L2 cache - if enabled). The new entity object is then added to the persistence context as a managed entity object and returned to the application.

Notice that construction of a new managed object during retrieval uses the no-arg constructor. Therefore, it is recommended to avoid unnecessary time consuming operations in no-arg constructors of entity classes and to keep them simple as possible.

Retrieval by Class and Primary Key

Every entity object can be uniquely identified and retrieved by the combination of its class and its primary key. Given an `EntityManager em`, the following code fragment demonstrates retrieval of an `Employee` object whose primary key is 1:

```
Employee employee = em.find(Employee.class, 1);
```

Casting of the retrieved object to `Employee` is not required because `find` is defined as returning an

instance of the same class that it takes as a first argument (using generics).
An `IllegalArgumentException` is thrown if the specified class is not an entity class.

If the `EntityManager` already manages the specified entity object in its persistence context no retrieval is required and the existing managed object is returned as is. Otherwise, the object data is retrieved from the database and a new managed entity object with that retrieved data is constructed and returned. If the object is not found in the database `null` is returned.

A similar method, `getReference`, can be considered the lazy version of `find`:

```
Employee employee = em.getReference(Employee.class, 1);
```

The `getReference` method works like the `find` method except that if the entity object is not already managed by the `EntityManager` a *hollow* object might be returned (`null` is never returned). A hollow object is initialized with the valid primary key but all its other persistent fields are uninitialized. The object content is retrieved from the database and the persistent fields are initialized, lazily, when the entity object is first accessed. If the requested object does not exist an `EntityNotFoundException` is thrown when the object is first accessed.

The `getReference` method is useful when a reference to an entity object is required but not its content, such as when a reference to it has to be set from another entity object.

Retrieval by Eager Fetch

Retrieval of an entity object from the database might cause automatic retrieval of additional entity objects. By default, a retrieval operation is automatically cascaded through all the non collection and map persistent fields (i.e. through one-to-one and many-to-one relationships). Therefore, when an entity object is retrieved, all the entity objects that are reachable from it by navigation through non collection and map persistent fields are also retrieved. Theoretically, in some extreme situations this might cause the retrieval of the entire database into the memory, which is usually unacceptable.

A persistent reference field can be excluded from this automatic cascaded retrieval by using a lazy fetch type:

```
@Entity
class Employee {
   :
  @ManyToOne(fetch=FetchType.LAZY)
  private Employee manager;
   :
```

```
}
```

The default for non collection and map references is `FetchType.EAGER`, indicating that the retrieval operation is cascaded through the field. Explicitly specifying `FetchType.LAZY` in either `@OneToOne` or `@ManyToOne` annotations (currently ObjectDB does not distinguish between the two) excludes the field from participating in retrieval cascading.

When an entity object is retrieved all its persistent fields are initialized. A persistent reference field with the `FetchType.LAZY` fetch policy is initialized to reference a new managed hollow object (unless the referenced object is already managed by the `EntityManager`). In the example above, when an `Employee` instance is retrieved its `manager` field might reference a hollow `Employee` instance. In a hollow object the primary key is set but other persistent fields are uninitialized until the object fields are accessed.

On the other hand, the default fetch policy of persistent collection and map fields is `FetchType.LAZY`. Therefore, by default, when an entity object is retrieved any other entity objects that it references through its collection and map fields are not retrieved with it.

This can be changed by an explicit `FetchType.EAGER` setting:

```
@Entity
class Employee {
   :
  @ManyToMany(fetch=FetchType.EAGER)
  private Collection<Project> projects;
   :
}
```

Specifying `FetchType.EAGER` explicitly in `@OneToMany` or `@ManyToMany` annotations (currently ObjectDB does not distinguish between the two) enables cascading retrieval for the field. In the above example, when an `Employee` instance is retrieved all the referenced `Project` instances are also retrieved automatically.

Retrieval by Navigation and Access

All the persistent fields of an entity object can be accessed freely, regardless of the current fetch policy, as long as the `EntityManager` is open. This also includes fields that reference entity objects that have not been loaded from the database yet and are represented by hollow objects. If the `EntityManager` is open when a hollow object is first accessed its content is automatically retrieved from the database and all its persistent fields are initialized.

From the point of view of the developer it looks like the entire graph of objects is present in memory. This

illusion, which is based on lazy transparent activation and retrieval of objects by ObjectDB, helps hide some of the direct interaction with the database and makes database programming easier.

For example, after retrieving an `Employee` instance from the database the `manager` field may include a hollow `Employee` entity object:

```
Employee employee = em.find(Employee.class, 1);
Employee managed = employee.getManager(); // might be hollow
```

If `manager` is hollow transparent activation occurs when it is first accessed. For example:

```
String managerName = manager.getName();
```

Accessing a persistent field in a hollow object (e.g. the name of the `manager` in the example above) causes the retrieval of missing content from the database and initialization of all the persistent fields.

As seen, the entire graph of objects is available for navigation, regardless of the fetch policy. The fetch policy, however, does affect performance. Eager retrieval might minimize the round trips to the database and improve performance, but unnecessary retrieval of entity objects that are not in use will decrease performance.

The fetch policy also affects objects that become detached (e.g. when the `EntityManager` is closed). Transparent activation is not supported for detached objects. Therefore, only content that has already been fetched from the database is available in objects that are detached.

JPA 2 introduces methods for checking if a specified entity object or a specified persistent field is loaded. For example:

```
PersistenceUtil util = Persistence.getPersistenceUtil();
boolean isObjectLoaded = util.isLoaded(employee);
boolean isFieldLoaded = util.isLoaded(employee, "address");
```

As shown above, a `PersistenceUtil` instance is obtained from the static `getPersistenceUtil` method. It provides two `isLoaded` methods - one for checking an entity object and the other for checking a persistent field of an entity object.

Retrieval by Query

The most flexible method for retrieving objects from the database is to use queries. The official query

language of JPA is JPQL (Java Persistence Query Language). It enables retrieval of objects from the database by using simple queries as well as complex, sophisticated ones. JPA queries and JPQL are described in chapter 4.

Retrieval by Refresh

Managed objects can be reloaded from the database by using the `refresh` method:

```
em.refresh(employee);
```

The content of the managed object in memory is discarded (including changes, if any) and replaced by data that is retrieved from the database. This might be useful to ensure that the application deals with the most up to date version of an entity object, just in case it might have been changed by another `EntityManager` since it was retrieved.

An `IllegalArgumentException` is thrown by `refresh` if the argument is not a managed entity (including entity objects in the New, Removed or Detached states). If the object does not exist in the database anymore an `EntityNotFoundException` is thrown.

Cascading Refresh

Marking a reference field with `CascadeType.REFRESH` (or `CascadeType.ALL`, which includes REFRESH) indicates that `refresh` operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.<span class="code">REFRESH</span>)
    private Address address;
     :
}
```

In the example above, the `Employee` entity class contains an `address` field that references an instance of `Address`, which is another entity class. Due to the `CascadeType.REFRESH` setting, when an `Employee` instance is refreshed the operation is automatically cascaded to the referenced `Address` instance, which is then automatically refreshed as well. Cascading may continue recursively when applicable (e.g. to entity objects that the `Address` object references, if any).

3.3.3 Updating JPA Entity Objects

Modifying existing entity objects that are stored in the database is based on transparent persistence, which means that changes are detected and handled automatically.

Transparent Update

Once an entity object is retrieved from the database (no matter which way) it can simply be modified in memory from inside an active transaction:

```
Employee employee = em.find(Employee.class, 1);

em.getTransaction().begin();
employee.setNickname("Joe the Plumber");
em.getTransaction().commit();
```

The entity object is physically updated in the database when the transaction is committed. If the transaction is rolled back and not committed the update is discarded.

On commit the persist operation can be cascaded from all the entity objects that have to be stored in the database, including from all the modified entity objects. Therefore, entity objects that are referenced from modified entity objects by fields that are marked with `CascadeType.PERSIST` or `CascadeType.ALL` are also persisted. If global cascade persist is enabled all the reachable entity objects that are not managed yet are also persisted.

Automatic Change Tracking

As shown above, an update is achieved by modifying a managed entity object from within an active transaction. No `EntityManager`'s method is invoked to report the update. Therefore, to be able to apply database updates on commit, ObjectDB must detect changes to managed entities automatically. One way to detect changes is to keep a snapshot of every managed object when it is retrieved from the database and to compare that snapshot to the actual managed object on commit. A more efficient way to detect changes automatically is described in the Enhancer section in chapter 5.

However, detecting changes to arrays requires using snapshots even if the entity classes are enhanced. Therefore, for efficiency purposes, the default behavior of ObjectDB ignores array changes when using enhanced entity classes:

```
Employee employee = em.find(Employee.class, 1);
```

```
em.getTransaction().begin();
employee.projects[0] = new Project(); // not detected automatically
JDOHelper.makeDirty(employee, "projects"); // reported as dirty
em.getTransaction().commit();
```

As demonstrated above, array changes are not detected automatically (by default) but it is possible to report a change explicitly by invoking the JDO's makeDirty method.

Alternatively, ObjectDB can be configured to detect array changes using snapshots as well as when enhanced entity classes are in use.

It is usually recommended to use collections rather than arrays when using JPA. Collections are more portable to ORM JPA implementations and provide better automatic change tracking support.

3.3.4 Deleting JPA Entity Objects

Existing entity objects can be deleted from the database either explicitly by invoking the `remove` method or implicitly as a result of a cascade operation.

Explicit Remove

In order to delete an object from the database it has ?to first be retrieved (no matter which way) and then in an active transaction, it can be deleted using the `remove` method:

```
Employee employee = em.find(Employee.class, 1);

em.getTransaction().begin();
em.remove(employee);
em.getTransaction().commit();
```

The entity object is physically deleted from the database when the transaction is committed. Embedded objects that are contained in the entity object are also deleted. If the transaction is rolled back and not committed the object is not deleted.

An `IllegalArgumentException` is thrown by `remove` if the argument is not a an instance of an entity class or if it is a detached entity. A `TransactionRequiredException` is thrown if there is no active transaction when `remove` is called because operations that modify the database require an active transaction.

Cascading Remove

Marking a reference field with `CascadeType.REMOVE` (or `CascadeType.ALL`, which includes `REMOVE`) indicates that `remove` operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.REMOVE)
    private Address address;
     :
}
```

In the example above, the `Employee` entity class contains an `address` field that references an instance of `Address`, which is another entity class. Due to the `CascadeType.REMOVE` setting, when an `Employee` instance is removed the operation is automatically cascaded to the referenced `Address` instance, which is then automatically removed as well. Cascading may continue recursively when applicable (e.g. to entity objects that the `Address` object references, if any).

Orphan Removal

JPA 2 supports an additional and more aggressive remove cascading mode which can be specified using the `orphanRemoval` element of the `@OneToOne` and `@OneToMany` annotations:

```
@Entity
class Employee {
     :
    @OneToOne(orphanRemoval=true)
    private Address address;
     :
}
```

When an `Employee` entity object is removed the remove operation is cascaded to the referenced `Address` entity object. In this regard, `orphanRemoval=true` and `cascade=CascadeType.REMOVE` are identical, and if `orphanRemoval=true` is specified, `CascadeType.REMOVE` is redundant.

The difference between the two settings is in the response to disconnecting a relationship. For example, such as when setting the `address` field to `null` or to another `Address` object.

- If `orphanRemoval=true` is specified the disconnected `Address` instance is automatically removed. This is useful for cleaning up dependent objects (e.g. `Address`) that should not exist without a reference from an owner object (e.g. `Employee`).
- If only `cascade=CascadeType.REMOVE` is specified no automatic action is taken since disconnecting a relationship is not a remove operation.

To avoid dangling references as a result of orphan removal this feature should only be enabled for fields that hold private non shared dependent objects.

Orphan removal can also be set for collection and map fields. For example:

```
@Entity
class Employee {
     :
    @OneToMany(orphanRemoval=true)
    private List<Address> addresses;
     :
}
```

In this case, removal of an `Address` object from the collection leads to automatic removal of that object from the database.

3.4 Advanced JPA Topics

This section discusses advanced JPA topics:

- Detached Entity Objects
- Locking in JPA
- JPA Lifecycle Events
- Shared (L2) Entity Cache
- JPA Metamodel API

3.4.1 Detached Entity Objects

Detached entity objects are objects in a special state in which they are not managed by any `EntityManager` but still represent objects in the database. Compared to managed entity objects, detached objects are limited in functionality:

- Many JPA methods do not accept detached objects (e.g. `lock`).
- Retrieval by navigation from detached objects is not supported, so only persistent fields that have been loaded before detachment should be used.
- Changes to detached entity objects are not stored in the database unless modified detached objects are merged back into an `EntityManager` to become managed again.

Detached objects are useful in situations in which an `EntityManager` is not available and for transferring objects between different `EntityManager` instances.

Explicit Detach

When a managed entity object is serialized and then deserialized, the deserialized entity object (but not the original serialized object) is constructed as a detached entity object since is not associated with any `EntityManager`.

In addition, in JPA 2 we can detach an entity object by using the `detach` method:

```
em.detach(employee);
```

An `IllegalArgumentException` is thrown by `detach` if the argument is not an entity object.

Cascading Detach

Marking a reference field with `CascadeType.DETACH` (or `CascadeType.ALL`, which includes `DETACH`) indicates that `detach` operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.DETACH)
    private Address address;
     :
}
```

In the example above, the `Employee` entity class contains an `address` field that references an instance of `Address`, which is another entity class. Due to the `CascadeType.DETACH` setting, when an `Employee` instance is detached the operation is automatically cascaded to the referenced `Address` instance, which is then automatically detached as well. Cascading may continue recursively when applicable (e.g. to entity

objects that the Address object references, if any).

Bulk Detach

The following operations clear the entire EntityManager's persistence context and detach all managed entity objects:

- Invocation of the close method, which closes an EntityManager.
- Invocation of the clear method, which clears an EntityManager's persistence context.
- Rolling back a transaction - either by invocation of rollback or by a commit failure.

Explicit Merge

Detached objects can be attached to any EntityManager by using the merge method:

```
em.merge(employee);
```

The content of the specified detached entity object is copied into an existing managed entity object with the same identity (i.e. same type and primary key). If the EntityManager does not manage such an entity object yet a new managed entity object is constructed. The detached object itself, however, remains unchanged and detached.

An IllegalArgumentException is thrown by merge if the argument is not an instance of an entity class or it is a removed entity. A TransactionRequiredException is thrown if there is no active transaction when merge is called because operations that might modify the database require an active transaction.

Cascading Merge

Marking a reference field with CascadeType.MERGE (or CascadeType.ALL, which includes MERGE) indicates that merge operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.MERGE)
    private Address address;
     :
```

```
}
```

In the example above, the `Employee` entity class contains an `address` field that references an instance of `Address`, which is another entity class. Due to the `CascadeType.MERGE` setting, when an `Employee` instance is merged the operation is automatically cascaded to the referenced `Address` instance, which is then automatically merged as well. Cascading may continue recursively when applicable (e.g. to entity objects that the `Address` object references, if any).

3.4.2 Locking in JPA

JPA 2 supports both **optimistic locking** and **pessimistic locking**. Locking is essential to avoid update collisions resulting from simultaneous updates to the same data by two concurrent users. Locking in ObjectDB (and in JPA) is always at the database object level, i.e. each database object is locked separately.

Optimistic locking is applied on transaction commit. Any database object that has to be updated or deleted is checked. An exception is thrown if it is found out that an update is being performed on an old version of a database object, for which another update has already been committed by another transaction.

When using ObjectDB, optimistic locking is enabled by default and fully automatic. Optimistic locking should be the first choice for most applications, since compared to pessimistic locking it is easier to use and more efficient.

In the rare cases in which update collision must be revealed earlier (before transaction commit) **pessimistic locking** can be used. When using pessimistic locking, database objects are locked during the transaction and lock conflicts, if they happen, are detected earlier.

Optimistic Locking

ObjectDB maintains a version number for every entity object. The initial version of a new entity object (when it is stored in the database for the first time) is 1. In every transaction in which an entity object is modified its version number is automatically increased by one. Version numbers are managed internally but can be exposed by defining a version field.

During `commit` (and `flush`), ObjectDB checks every database object that has to be updated or deleted, and compares the version number of that object in the database to the version number of the in-memory object being updated. The transaction fails and an `OptimisticLockException` is thrown if the version numbers do not match, indicating that the object has been modified by another user (using another `EntityManager`) since it was retrieved by the current updater.

Optimistic locking is completely automatic and enabled by default in ObjectDB, regardless if a version field (which is required by some ORM JPA providers) is defined in the entity class or not.

Pessimistic Locking

The main supported pessimistic lock modes are:

- PESSIMISTIC_READ - which represents a shared lock.
- PESSIMISTIC_WRITE - which represents an exclusive lock.

Setting a Pessimistic Lock

An entity object can be locked explicitly by the lock method:

```
em.lock(employee, LockModeType.<span class="code">PESSIMISTIC_WRITE</span>);
```

The first argument is an entity object. The second argument is the requested lock mode.

A TransactionRequiredException is thrown if there is no active transaction when lock is called because explicit locking requires an active transaction.

A LockTimeoutException is thrown if the requested pessimistic lock cannot be granted:

- A PESSIMISTIC_READ lock request fails if another user (which is represented by another EntityManager instance) currently holds a PESSIMISTIC_WRITE lock on that database object.
- A PESSIMISTIC_WRITE lock request fails if another user currently holds either a PESSIMISTIC_WRITE lock or a PESSIMISTIC_READ lock on that database object.

For example, consider the following code fragment:

```
em1.lock(e1, lockMode1);
em2.lock(e2, lockMode2);
```

em1 and em2 are two EntityManager instances that manage the same Employee database object, which is referenced as e1 by em1 and as e2 by em2 (notice that e1 and e2 are two in-memory entity objects that represent one database object).

If both lockMode1 and lockMode2 are PESSIMISTIC_READ - these lock requests should succeed. Any other combination of pessimistic lock modes, which also includes PESSIMISTIC_WRITE, will cause a LockTimeoutException (on the second lock request).

Pessimistic Lock Timeout

By default, when a pessimistic lock conflict occurs a `LockTimeoutException` is thrown immediately. The `"javax.persistence.lock.timeout"` hint can be set to allow waiting for a pessimistic lock for a specified number of milliseconds. The hint can be set in several scopes:

For the entire persistence unit - using a `persistence.xml` property:

```
<properties>
   <property name="javax.persistence.lock.timeout" value="1000"/>
</properties>
```

For an `EntityManagerFactory` - using the `createEntityManagerFacotory` method:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.lock.timeout", 2000);
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("pu", properties);
```

For an `EntityManager` - using the `createEntityManager` method:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.lock.timeout", 3000);
EntityManager em = emf.createEntityManager(properties);
```

or using the `setProperty` method:

```
em.setProperty("javax.persistence.lock.timeout", 4000);
```

In addition, the hint can be set for a specific retrieval operation or query.

Releasing a Pessimistic Lock

Pessimistic locks are automatically released at transaction end (using either `commit` or `rollback`).

ObjectDB supports also releasing a lock explicitly while the transaction is active, as so:

```

```

```
em.lock(employee, LockModeType.NONE);
```

Other Explicit Lock Modes

In addition to the two main pessimistic modes (`PESSIMISTIC_WRITE` and `PESSIMISTIC_READ`, which are discussed above), JPA defines additional lock modes that can also be specified as arguments for the `lock` method to obtain special effects:

- `OPTIMISTIC` (formerly `READ`)
- `OPTIMISTIC_FORCE_INCREMENT` (formerly `WRITE`)
- `PESSIMISTIC_FORCE_INCREMENT`

Since optimistic locking is applied automatically by ObjectDB to every entity object, the `OPTIMISTIC` lock mode has no effect and, if specified, is silently ignored by ObjectDB.

The `OPTIMISTIC_FORCE_INCREMENT` mode affects only clean (non dirty) entity objects. Explicit lock at that mode marks the clean entity object as modified (dirty) and increases its version number by 1.

The `PESSIMISTIC_FORCE_INCREMENT` mode is equivalent to the `PESSIMISTIC_WRITE` mode with the addition that it marks a clean entity object as dirty and increases its version number by one (i.e. it combines `PESSIMISTIC_WRITE` with `OPTIMISTIC_FORCE_INCREMENT`).

Locking during Retrieval

JPA 2 provides various methods for locking entity objects when they are retrieved from the database. In addition to improving efficiency (relative to a retrieval followed by a separate lock), these methods perform retrieval and locking as one atomic operation.

For example, the `find` method has a form that accepts a lock mode:

```
Employee employee = em.find(
    Employee.class, 1, LockModeType.<span
class="code">PESSIMISTIC_WRITE</span>);
```

Similarly, the `refresh` method can also receive a lock mode:

```
em.refresh(employee, LockModeType.<span
class="code">PESSIMISTIC_WRITE</span>);
```

A lock mode can also be set for a query in order to lock all the query result objects.

When a retrieval operation includes pessimistic locking, timeout can be specified as a property. For example:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.lock.timeout", 2000);

Employee employee = em.find(
    Employee.class, 1, LockModeType.PESSIMISTIC_WRITE, properties);

...

em.refresh(employee, LockModeType.PESSIMISTIC_WRITE, properties);
```

Setting timeout at the operation level overrides setting in higher scopes.

3.4.3 JPA Lifecycle Events

Callback methods are user defined methods that are attached to entity lifecycle events and are invoked automatically by JPA when these events occur.

Internal Callback Methods

Internal callback methods are methods that are defined within an entity class. For example, the following entity class defines all the supported callback methods with empty implementations:

```
@Entity
public static class MyEntityWithCallbacks {
    @PrePersist void onPrePersist() {}
    @PostPersist void onPostPersist() {}
    @PostLoad void onPostLoad() {}
    @PreUpdate void onPreUpdate() {}
    @PostUpdate void onPostUpdate() {}
    @PreRemove void onPreRemove() {}
    @PostRemove void onPostRemove() {}
}
```

Internal callback methods should always return `void` and take no arguments. They can have any name and any access level (`public`, `protected`, `package` and `private`) but should not be `static`.

The annotation specifies when the callback method is invoked:

- `@PrePersist` - before a new entity is persisted (added to the `EntityManager`).
- `@PostPersist` - after storing a new entity in the database (during `commit` or `flush`).
- `@PostLoad` - after an entity has been retrieved from the database.
- `@PreUpdate` - when an entity is identified as modified by the `EntityManager`.
- `@PostUpdate` - after updating an entity in the database (during `commit` or `flush`).
- `@PreRemove` - when an entity is marked for removal in the EntityManager.
- `@PostRemove` - after deleting an entity from the database (during `commit` or `flush`).

An entity class may include callback methods for any subset or combination of lifecycle events but no more than one callback method for the same event. However, the same method may be used for multiple callback events by marking it with more than one annotation.

By default, a callback method in a super entity class is also invoked for entity objects of the subclasses unless that callback method is overridden by the subclass.

Implementation Restrictions

To avoid conflicts with the original database operation that fires the entity lifecycle event (which is still in progress) callback methods should not call `EntityManager` or `Query` methods and should not access any other entity objects.

If a callback method throws an exception within an active transaction, the transaction is marked for rollback and no more callback methods are invoked for that operation.

Listeners and External Callback Methods

External callback methods are defined outside entity classes in a special listener class:

```
public class MyListener {
    @PrePersist void onPrePersist(Object o) {}
    @PostPersist void onPostPersist(Object o) {}
    @PostLoad void onPostLoad(Object o) {}
    @PreUpdate void onPreUpdate(Object o) {}
    @PostUpdate void onPostUpdate(Object o) {}
    @PreRemove void onPreRemove(Object o) {}
    @PostRemove void onPostRemove(Object o) {}
}
```

External callback methods (in a listener class) should always return `void` and take one argument that

specifies the entity which is the source of the lifecycle event. The argument can have any type that matches the actual value (e.g. in the code above, `Object` can be replaced by a more specific type). The listener class should be stateless and should have a public no-arg constructor (or no constructor at all) to enable automatic instantiation.

The listener class is attached to the entity class using the `@EntityListeners` annotation:

```
@Entity @EntityListeners(MyListener.class)
public class MyEntityWithListener {
}
```

Multiple listener classes can also be attached to one entity class:

```
@Entity @EntityListeners({MyListener1.class, MyListener2.class})
public class MyEntityWithTwoListeners {
}
```

Listeners that are attached to an entity class are inherited by its subclasses unless the subclass excludes inheritance explicitly using the `@ExcludeSuperclassListeners` annotation:

```
@Entity @ExcludeSuperclassListeners
public class EntityWithNoListener extends EntityWithListener {
}
```

Default Entity Listeners

Default entity listeners are listeners that should be applied by default to all the entity classes. Currently, default listeners can only be specified in a mapping XML file because there is no equivalent annotation:

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="samples.MyDefaultListener1" />
        <entity-listener class="samples.MyDefaultListener2" />
      </entity-listeners>
```

```
    </persistence-unit-defaults>
  </persistence-unit-metadata>
</entity-mappings>
```

The mapping file has to be located either in the default location, `META-INF/orm.xml`, or in another location that is specified explicitly in the persistence unit definition (in `persistence.xml`).

Default listeners are applied by default to all the entity classes. The `@ExcludeDefaultListeners` annotation can be used to exclude an entity class and all its descendant classes from using the default listeners:

```
@Entity @ExcludeDefaultListeners
public class NoDefaultListenersForThisEntity {
}

@Entity
public class NoDefaultListenersForThisEntityEither
  extends NoDefaultListenersForThisEntity {
}
```

Callback Invocation Order

If more than one callback method has to be invoked for a lifecycle event (e.g. from multiple listeners) the invocation order is based on the following rules:

- All the external callback methods (which are defined in listeners) are invoked before the internal callback methods (which are defined in entity classes).
- Default listeners are handled first, then listeners of the top level entity class, and then down the hierarchy until listeners of the actual entity class. If there is more than one default listener or more than one listener at the same level in the hierarchy, the invocation order follows the definition order.
- Internal callback methods are invoked starting at the top level entity class and then down the hierarchy until the callback methods in the actual entity class are invoked.

3.4.4 Shared (L2) Entity Cache

Every `EntityManager` owns a persistence context, which is a collection of all the entity objects that it manages. The persistence context serves as a first level cache. An attempt to retrieve an entity object that is already managed by the `EntityManager` returns the existing instance from the persistence context,

rather than a new instantiated entity object.

The scope of the persistence context is one `EntityManager`. This section describes a level 2 (L2) cache of entity objects, which is managed by the `EntityManagerFactory` and shared by all its `EntityManager` objects. the broader scope of this cache makes it useful in applications that use many short term `EntityManager` instances.

In addition to the `EntityManager`'s L1 cache and the `EntityManagerFactory`'s L2 cache, which are managed on the client side - ObjectDB manages also several caches on the server side:

- Cache of database file pages.
- Cache of query programs.
- Cache of query execution results.

The scope of these server side caches is wider, since they exist per database and are shared by all the `EntityManagerFactory` and `EntityManager` instances of the same database - including on different client machines.

Setting the Shared Cache

The shared (L2) cache is configured in three scopes:

- Globally in the ObjectDB configuration.
- Per persistence unit in the `persistence.xml` file.
- Per entity class - using annotations.

ObjectDB Configuration

The shared cache size is specified in the ObjectDB configuration:

```
    <cache ... level2="0mb" />
```

The `level2` attribute determines the size of the `EntityManagerFactory`'s shared cache. The default size, `0`, indicates that the cache is disabled. To enable the cache a positive value has to be specified.

Persistence Unit Settings

The shared cache can also be enabled or disabled using a persistence unit property:

```
   <persistence-unit name="my-pu">
```

```
    ...
    <properties>
      <property name="javax.persistence.sharedCache.mode" value="ALL"/>
    </properties>
    ...
  </persistence-unit>
```

The `javax.persistence.sharedCache.mode` property can be set to one of the following values:

- `NONE` - cache is disabled.
- `ENABLE_SELECTIVE` - cache is disabled except for selected entity classes (see below).
- `DISABLE_SELECTIVE` - cache is enabled except for selected entity classes (see below).
- `ALL` (the default) - cache is enabled for all the entity classes.
- `UNSPECIFIED` - handled differently by different JPA providers. In ObjectDB the `UNSPECIFIED` value is equivalent to `ALL`, which is the default.

If the cache size is `0` - the shared cache is disabled regardless of the set mode.

Entity Class Cache Settings

The `ENABLE_SELECTIVE` mode indicates that the cache is disabled for all the entity classes except classes that are specified as `Cacheable` explicitly. For example:

```
@Cacheable // or @Cacheable(true)
@Entity
public class MyCacheableEntityClass {
    ...
}
```

Similarly, the `DISABLE_SELECTIVE` value indicates that the cache is enabled for all the entity classes except classes that are specified as non `Cacheable` explicitly. For example:

```
@Cacheable(false)
@Entity
public class MyNonCacheableEntityClass extends MyCacheableEntityClass {
    ...
}
```

`Cacheable` is an inherited property - every entity class which is not marked with `@Cacheable` inherits cacheability setting from its super class.

Using the Shared Cache

The shared cache (when enabled) provides the following functionality automatically:

- **On retrieval** - shared cache is used for entity objects that are not in the persistence context. If an entity object is not available also in the shared cache - it is retrieved from the database and added to the shared cache.
- **On commit** - new and modified entity objects are added to the shared cache.

JPA provides two properties that can be used in order to change the default behavior.

javax.persistence.cache.retrieveMode

The `"javax.persistence.cache.retrieveMode"` property specifies if the shared cache is used on retrieval. Two values are available for this property as constants of the `CacheRetrieveMode` enum:

- `CacheRetrieveMode.USE` - cache is used.
- `CacheRetrieveMode.BYPASS` - cache is not used.

The default setting is `USE`. It can be changed for a specific `EntityManager`:

```
  em.setProperty(
      "javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS);
```

Setting can also be overridden for a specific retrieval operation:

```
  // Before executing a query:
  query.setHint("javax.persistence.cache.retrieveMode",
CacheRetrieveMode.BYPASS);

  // For retrieval by type and primary key:
  em.find(MyEntity2.class, Long.valueOf(1),
      Collections.<String,Object>singletonMap(
        "javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS));
```

javax.persistence.cache.storeMode

The `"javax.persistence.cache.storeMode"` property specifies if new data should be added to the cache on commit and on retrieval. The property has three valid values, which are defined as constants of the `CacheStoreMode` enum:

- `CacheStoreMode.BYPASS` - cache is not updated with new data.
- `CacheStoreMode.USE` - new data in stored in the cache - but only for entity objects that are not in the cache already.
- `CacheStoreMode.REFRESH` - new data is stored in the cache - refreshing entity objects that are already cached.

The default setting is USE. It can be changed for a specific `EntityManager`:

```
em.setProperty("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS);
```

Setting can also be overridden for a specific retrieval operation. For example:

```
em.find(MyEntity2.class, Long.valueOf(1),
    Collections.<String,Object>singletonMap(
        "javax.persistence.cache.storeMode", CacheRetrieveMode.BYPASS));
```

The difference between `CacheStoreMode.USE` and `CacheStoreMode.REFRESH` is when bypassing the cache in retrieval operations. In this case, an entity object that is already cached is updated using the fresh retrieved data only when `CacheStoreMode.REFRESH` is used. This might be useful when the database might be updated by other applications (or using other `EntityManagerFactory` instances).

Using the Cache Interface

The shared cache is represented by the `Cache` interface. A `Cache` instance can be obtained by using the `EntityManagerFactory's getCache method`:

```
Cache cache = emf.getCache();
```

The `Cache` object enables checking if a specified entity object is cached:

```
boolean isCached = cache.contains(MyEntity.class, Long.valueOf(id));
```

Cached entity objects can be removed from the cache by one of the `evict` methods:

```
// Remove a specific entity object from the shared cache:
cache.evict(MyEntity.class, Long.valueOf(id));
```

```
// Remove all the instances of a specific class from the cache:
cache.evict(MyEntity.class);

// Clear the shared cache by removing all the cached entity objects:
cache.evictAll();
```

The Cache interface and its methods are unnecessary in most applications.

3.4.5 JPA Metamodel API

The JPA Metamodel API provides the ability to examine the persistent object model and retrieve details on managed classes and persistent fields and properties, similarly to the ability that Java reflection provides for general Java types.

The Metamodel Interface

The main interface of the JPA Metamodel API is `Metamodel`. It can be obtained either by the `EntityManagerFactory`'s `getMetamodel` method or by the `EntityManager`'s `getMetamodel` method (both methods are equivalent).

For example, given an `EntityManager`, `em`, a `Metamodel` instance can be obtained by:

```
Metamodel metamodel = em.getMetamodel();
```

The `Metamodel` interface provides several methods for exploring user defined persistable types (which are referred to as managed types) in the persistent object model.

Three methods can be used to retrieve sets of types:

```
// Get all the managed classes:
// (entity classes, embeddable classes, mapped super classes)
Set<ManagedType> allManagedTypes = metamodel.getManagedTypes();

// Get all the entity classes:
Set<EntityType> allEntityTypes = metamodel.getEntities();

// Get all the embeddable classes:
```

```
Set<EmbeddableType> allEmbeddableTypes = metamodel.getEmbeddables();
```

If managed classes are not listed in the persistence unit (which is optional when using ObjectDB) then only known managed types are returned. This includes all the types whose instances are already stored in the database.

Three additional methods can be used to retrieve a specific type by its `Class` instance:

```
// Get a managed type (entity, embeddable or mapped super classes):
ManagedType<MyClass> type1 = metamodel.managedType(MyClass.class);

// Get an entity type:
EntityType<MyEntity> type2 = metamodel.entity(MyEntity.class);

// Get an embeddable type:
EmbeddableType<MyEmbeddableType> type3 =
    metamodel.embeddable(MyEmbeddableType.class);
```

These three methods can also be used with types that are still unknown to ObjectDB (not listed in the persistence unit and have not been used yet). In this case, calling the method introduces the specified type to ObjectDB.

Type Interface Hierarchy

Types are represented in the Metamodel API by descendant interfaces of the `Type` interface:

- `BasicType` - represents system defined types.
- `ManagedType` is an ancestor of interfaces that represent user defined types:
  - `EmbeddableType` - represents user defined embeddable classes.
  - `IdentifiableType` is as a super interface of:
    - `MappedSuperclassType` - represents user defined mapped super classes.
    - `EntityType` - represents user defined entity classes.

The `Type` interfaces provides a thin wrapper of `Class` with only two methods:

```
// Get the underlying Java representation of the type:
Class cls = type.getJavaType();

// Get one of BASIC, EMBEDDABLE, ENTITY, MAPPED_SUPERCLASS:
```

```
PersistenceType kind = type.getPersistenceType();
```

The ManagedType interface adds methods for exploring managed fields and properties (which are referred to as attributes). For example:

```
// Get all the attributes - including inherited:
Set<Attribute> attributes1 = managedType.getAttributes();

// Get all the attributes - excluding inherited:
Set<Attribute> attributes2 = managedType.getDeclaredAttributes();

// Get a specific attribute - including inherited:
Attribute<MyClass,String> strAttr1 = managedType.getAttribute("name");

// Get a specific attribute - excluding inherited:
Attribute<MyClass,String> strAttr2 = managedType.getDeclaredAttribute("name");
```

Additional methods are defined in ManagedType to return attributes of Collection, List, Set a Map types in a type safe manner.

The IdentifiableType adds methods for retrieving information on the primary key and the version attributes and the super type. For example:

```
// Get the super type:
IdentifiableType<MyEntity> superType = entityType.getSupertype();

// Checks if the type has a single ID attribute:
boolean hasSingleId = entityType.hasSingleIdAttribute();

// Gets a single ID attribute - including inherited:
SingularAttribute<MyEntity,Long> id1 = entityType.getId(Long.class);

// Gets a single ID attribute - excluding inherited:
SingularAttribute<MyEntity,Long> id2 = entityType.getDeclaredId(Long.class);

// Checks if the type has a version attribute:
boolean hasVersion = entityType.hasVersionAttribute();

// Gets the version attribute - excluding inherited:
SingularAttribute<MyEntity,Long> v1 = entityType.getVersion(Long.class);

// Gets the version attribute - including inherited:
```

```
  SingularAttribute<MyEntity,Long> v2 =
entityType.getDeclaredVersion(Long.class);
```

Additional methods are defined in `IdentifiableType` to support an ID class when using multiple ID fields or properties.

Finally, the `EntityType` interface adds only one additional method for getting the entity name:

```
String entityName = entityType.getName();
```

Attribute Interface Hierarchy

Managed fields and properties are represented by the `Attribute` interfaces and its descendant interfaces:

- `SingularAttribute` - represents single value attributes.
- `PluralAttribute` is an ancestor of interfaces that represent multi value attributes:
  - `CollectionAttribute` - represents attributes of `Collection` types.
  - `SetAttribute` - represents attributes of `Set` types.
  - `ListAttribute` represents attributes of `List` types.
  - `MapAttribute` - represents attributes of `Map` types.

The `Attribute` interface provides methods for retrieving field and property details. For example:

```
// Get the field (or property) name:
String name = attr.getName();

// Get Java representation of the field (or property) type:
Class<Integer> attr.getJavaType();

// Get Java reflection representation of the field (or property) type:
Member member = attr.getJavaMember();

// Get the type in which this field (or property) is defined:
ManagedType<MyEntity> entityType = attr.getDeclaringType();
```

Few other methods are defined in `Attribute` and in `MapAttribute` to support additional details.

Chapter 4 - JPA Queries (JPQL / Criteria)

The JPA Query Language (JPQL) can be considered as an object oriented version of SQL. Users familiar with SQL should find JPQL very easy to learn and use. This chapter explains how to use JPQL as well as how to use the JPA Criteria API, which provides an alternative way for building queries in JPA, based on JPQL.

The first section describes the API that JPA provides for using dynamic and static (named) queries. It explains how to use the relevant interfaces, annotations, enums and methods, but does not provide specific details on the JPQL query language itself:

- JPA Query API

The Java Persistence Query Language (JPQL) is discussed in the next two sections. First, the structure of a JPQL query (and a criteria query) is explained by describing the main clauses of JPQL queries (SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY):

- JPA Query Structure (JPQL / Criteria)

Then the expressions that are used for building JPQL and criteria query clauses are explained:

- JPA Query Expressions (JPQL / Criteria)

ObjectDB also supports the Java Data Objects (JDO) Query Language (JDOQL), which is more Java oriented and is based on the syntax of Java. JDOQL is not covered in this manual (see chapter 7 in ObjectDB 1.0 manual for a description JDOQL).

4.1 JPA Query API

Queries are represented in JPA 2 by two interfaces - the old `Query` interface, which was the only interface available for representing queries in JPA 1, and the new `TypedQuery` interface that was introduced in JPA 2. The `TypedQuery` interface extends the `Query` interface.

In JPA 2 the `Query` interface should be used mainly when the query result type is unknown or when a query returns polymorphic results and the lowest known common denominator of all the result objects is `Object`. When a more specific result type is expected queries should usually use the `TypedQuery` interface. It is easier to run queries and process the query results in a type safe manner when using the `TypedQuery` interface.

Building Queries with createQuery

As with most other operations in JPA, using queries starts with an `EntityManager` (represented by `em` in

the following code snippets), which serves as a factory for both `Query` and `TypedQuery`:

```
Query q1 = em.createQuery("SELECT c FROM Country c");

TypedQuery<Country> q2 =
    em.createQuery("SELECT c FROM Country c", Country.class);
```

In the above code, the same JPQL query which retrieves all the `Country` objects in the database is represented by both `q1` and `q2`. When building a `TypedQuery` instance the expected result type has to be passed as an additional argument, as demonstrated for `q2`. Because, in this case, the result type is known (the query returns only `Country` objects), a `TypedQuery` is preferred.

There is another advantage of using TypedQuery in ObjectDB. In the context of the queries above, if there are no Country instances in the database yet and the `Country` class is unknown as a managed entity class - only the `TypedQuery` variant is valid because it introduces the `Country` class to ObjectDB.

Dynamic JPQL, Criteria API and Named Queries

Building queries by passing JPQL query strings directly to the `createQuery` method, as shown above, is referred to in JPA as dynamic query construction because the query string can be built dynamically at runtime.

The JPA Criteria API provides an alternative way for building dynamic queries, based on Java objects that represent query elements (replacing string based JPQL).

JPA also provides a way for building static queries, as named queries, using the `@NamedQuery` and `@NamedQueries` annotations. It is considered to be a good practice in JPA to prefer named queries over dynamic queries when possible.

Organization of this Section

The following pages explain how to define and execute queries in JPA:

- Running JPA Queries
- Query Parameters in JPA
- JPA Named Queries
- JPA Criteria API Queries
- Setting and Tuning of JPA Queries

In addition, the syntax of the JPA Query Language (JPQL) is described in:

- JPA Query Structure (JPQL / Criteria)
- JPA Query Expressions (JPQL / Criteria)

4.1.1 Running JPA Queries

The `Query` interface defines two methods for running SELECT queries:

- `Query.getSingleResult` - for use when exactly one result object is expected.
- `Query.getResultList` - for general use in any other case.

Similarly, the `TypedQuery` interface defines the following methods:

- `TypedQuery.getSingleResult` - for use when exactly one result object is expected.
- `TypedQuery.getResultList` - for general use in any other case.

In addition, the Query interface defines a method for running DELETE and UPDATE queries:

- `Query.executeUpdate` - for running only DELETE and UPDATE queries.

Ordinary Query Execution (with getResultList)

The following query retrieves all the Country objects in the database. Because multiple result objects are expected, the query should be run using the getResultList method:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Both `Query` and `TypedQuery` define a `getResultList` method, but the version of `Query` returns a result list of a raw type (non generic) instead of a parameterized (generic) type:

```
Query query = em.createQuery("SELECT c FROM Country c");
List results = query.getResultList();
```

An attempt to cast the above `results` to a parameterized type (`List<Country>`) will cause a compilation warning. If, however, the new `TypedQuery` interface is used casting is unnecessary and the warning is avoided.

The query result collection functions as any other ordinary Java collection. A result collection of a

parameterized type can be iterated easily using an enhanced for loop:

```
for (Country c : results) {
    System.out.println(c.getName());
}
```

Note that for merely printing the country names, a query that uses projection and retrieves country names directly instead of fully built Country instances would be more efficient.

Single Result Query Execution (with getSingleResult)

The `getResultList` method (which was discussed above) can also be used to run queries that return a single result object. In this case, the result object has to be extracted from the result collection after query execution (e.g. by `results.get(0)`). To eliminate this routine operation JPA provides an additional method, `getSingleResult`, as a more convenient method when exactly one result object is expected.

The following aggregate query always returns a single result object, which is a `Long` object reflecting the number of `Country` objects in the database:

```
TypedQuery<Long> query = em.createQuery(
    "SELECT COUNT(c) FROM Country c", Long.class);
long countryCount = query.getSingleResult();
```

Notice that when a query returns a single object it might be tempting to prefer Query over TypedQuery even when the result type is known because the casting of a single object is easy and the code is simple:

```
Query query = em.createQuery("SELECT COUNT(c) FROM Country c");
long countryCount = (Long)query.getSingleResult();
```

An aggregate COUNT query always returns one result, by definition. In other cases our expectation for a single object result might fail, depending on the database content. For example, the following query is expected to return a single Country object:

```
Query query = em.createQuery(
    "SELECT c FROM Country c WHERE c.name = &#39;Canada&#39;");
Country c = (Country)query.getSingleResult();
```

However, the correctness of this assumption depends on the content of the database. If the database contains multiple `Country` objects with the name 'Canada' (e.g. due to a bug) a NonUniqueResultException is thrown. On the other hand, if there are no results at all a NoResultException is thrown. Therefore, using getSingleResult requires some caution and if there is any chance that these exceptions might be thrown they have to be caught and handled.

DELETE and UPDATE Query Execution (with executeUpdate)

DELETE and UPDATE queries are executed using the `executeUpdate` method.

For example, the following query deletes all the `Country` instances:

```
int count = em.createQuery("DELETE FROM Country").executeUpdate();
```

and the following query resets the area field in all the Country instances to zero:

```
int count = em.createQuery("UPDATE Country SET area = 0").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.

On success - the executeUpdate method returns the number of objects that have been updated or deleted by the query.

The Query Structure section explains DELETE and UPDATE queries in more detail.

4.1.2 Query Parameters in JPA

Query parameters enable the definition of reusable queries. Such queries can be executed with different parameter values to retrieve different results. Running the same query multiple times with different parameter values (arguments) is more efficient than using a new query string for every query execution, because it eliminates the need for repeated query compilations.

Named Parameters (:name)

The following method retrieves a `Country` object from the database by its name:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
```

```
        "SELECT c FROM Country c WHERE c.name = :name", Country.class);
    return query.setParameter("name", name).getSingleResult();
  }
```

The WHERE clause reduces the query results to `Country` objects whose name field value is equal to `:name`, which is a parameter that serves as a placeholder for a real value. Before the query can be executed a parameter value has to be set using the `setParameter` method. The `setParameter` method supports method chaining (by returning the same `TypedQuery` instance on which it was invoked), so invocation of `getSingleResult` can be chained to the same expression.

Named parameters can be easily identified in a query string by their special form, which is a colon (:) followed by a valid JPQL identifier that serves as the parameter name. JPA does not provide an API for defining the parameters explicitly (except when using criteria API), so query parameters are defined implicitly by appearing in the query string. The parameter type is inferred by the context. In the above example, a comparison of `:name` to a field whose type is `String` indicates that the type of `:name` itself is `String`.

Queries can include multiple parameters and each parameter can have one or more occurrences in the query string. A query can be run only after setting values for all its parameters (in no matter in which order).

Ordinal Parameters (?index)

In addition to named parameter, whose form is `:name`, JPQL supports also ordinal parameter, whose form is `?index`. The following method is equivalent to the method above, except that an ordinal parameter replaces the named parameter:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = ?1", Country.class);
  return query.setParameter(1, name).getSingleResult();
}
```

The form of ordinal parameters is a question mark (?) followed by a positive int number. Besides the notation difference, named parameters and ordinal parameters are identical.

Named parameters can provide added value to the clarity of the query string (assuming that meaningful names are selected). Therefore, they are preferred over ordinal parameters.

Criteria Query Parameters

In a JPA query that is built by using the JPA Criteria API - parameters (as other query elements) are represented by objects (of type `ParameterExpression` or its super interface `Parameter`) rather than by names or numbers.

See the Parameters in Criteria Queries section for more details.

Parameters vs. Literals

Following is a third version of the same method. This time without parameters:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = '" + name + "'",
      Country.class);
  return query.getSingleResult();
}
```

Instead of using a parameter for the queried name the new method embeds the name as a `String` literal. There are a few drawbacks to using literals rather than parameters in queries.

First, the query is not reusable. Different literal values lead to different query strings and each query string requires its own query compilation, which is very inefficient. On the other hand, when using parameters, even if a new `TypedQuery` instance is constructed on every query execution, ObjectDB can identify repeating queries with the same query string and use a cached compiled query program, if available.

Second, embedding strings in queries is unsafe and can expose the application to JPQL injection attacks. Suppose that the name parameter is received as an input from the user and then embedded in the query string as is. Instead of a simple country name, a malicious user may provide JPQL expressions that change the query and may help in hacking the system.

In addition, parameters are more flexible and support elements that are unavailable as literals, such as entity objects.

API Parameter Methods

Over half of the methods in `Query` and `TypedQuery` deal with parameter handling. The `Query` interface defines 18 such methods, 9 of which are overridden in `TypedQuery`. That large number of methods is not typical to JPA, which generally excels in its thin and simple API.

There are 9 methods for setting parameters in a query, which is essential whenever using query parameters. In addition, there are 9 methods for extracting parameter values from a query. These get methods, which are new in JPA 2, are expected to be much less commonly used than the set methods.

Two set methods are demonstrated above - one for setting a named parameter and the other for setting an ordinal parameter. A third method is designated for setting a parameter in a Criteria API query. The reason for having nine set methods rather than just three is that JPA additionally provides three separate methods for setting `Date` parameters as well as three separate methods for setting `Calendar` parameters.

`Date` and `Calendar` parameter values require special methods in order to specify what they represent, such as a pure date, a pure time or a combination of date and time, as explained in detail in the Date and Time (Temporal) Types section.

For example, the following invocation passes a `Date` object as a pure date (no time):

```
query.setParameter("date", new java.util.Date(), TemporalType.DATE);
```

Since `TemporalType.Date` represents a pure date, the time part of the newly constructed `java.util.Date` instance is discarded. This is very useful in comparison against a specific date, when time should be ignored.

The get methods support different ways to extract parameters and their values from a query, including by name (for named parameter), by position (for ordinal parameters) by Parameter object (for Criteria API queries), each with or without an expected type. There is also a method for extracting all the parameters as a set (`getParameters`) and a method for checking if a specified parameter has a value (`isBound`). These methods are not required for running queries and are expected to be less commonly used.

4.1.3 JPA Named Queries

A named query is a statically defined query with a predefined unchangeable query string. Using named queries instead of dynamic queries may improve code organization by separating the JPQL query strings from the Java code. It also enforces the use of query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

@NamedQuery and @NamedQueries Annotations

The following `@NamedQuery` annotation defines a query whose name is `"Country.findAll"` that retrieves all the `Country` objects in the database:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
```

The `@NamedQuery` annotation contains four elements - two of which are required and two are optional. The two required elements, `name` and `query` define the name of the query and the query string itself and are demonstrated above. The two optional elements, `lockMode` and `hints`, provide static replacement for the `setLockMode` and `setHint` methods.

Every `@NamedQuery` annotation is attached to exactly one entity class or mapped superclass - usually to the most relevant entity class. But since the scope of named queries is the entire persistence unit, names should be selected carefully to avoid collision (e.g. by using the unique entity name as a prefix).

It makes sense to add the above `@NamedQuery` to the `Country` entity class:

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
  ...
}
```

Attaching multiple named queries to the same entity class requires wrapping them in a `@NamedQueries` annotation, as follows:

```
@Entity
@NamedQueries({
    @NamedQuery(name="Country.findAll",
                query="SELECT c FROM Country c"),
    @NamedQuery(name="Country.findByName",
                query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
  ...
}
```

Note: Named queries can be defined in JPA XML mapping files instead of using the `@NamedQuery` annotation. ObjectDB supports JPA XML mapping files, including the definition of named queries. But, because mapping files are useful mainly for Object Relational Mapping (ORM) JPA providers and less so when using ObjectDB, this alternative is not covered in this manual.

Using Named Queries at Runtime

Named queries are represented at runtime by the same `Query` and `TypedQuery` interfaces but different `EntityManager` factory methods are used to instantiate them. The `createNamedQuery` method receives a query name and a result type and returns a `TypedQuery` instance:

```
TypedQuery<Country> query =
    em.createNamedQuery("Country.findAll", Country.class);
List<Country> results = query.getResultList();
```

Another form of `createNamedQuery` receives a query name and returns a `Query` instance:

```
Query query = em.createNamedQuery("SELECT c FROM Country c");
List results = query.getResultList();
```

One of the reasons that JPA requires the listing of managed classes in a persistence unit definition is to support named queries. Notice that named queries may be attached to any entity class or mapped superclass. Therefore, to be able to always locate any named query at runtime a list of all these managed persistable classes must be available.

ObjectDB makes the definition of a persistence unit optional. Named queries are automatically searched for in all the managed classes that ObjectDB is aware of, and that includes all the entity classes that have objects in the database. However, an attempt to use a named query still might fail if that named query is defined on a class that is still unknown to ObjectDB.

As a workaround, you may introduce classes to ObjectDB before accessing named queries, by using the JPA 2 `Metamodel` interface. For example:

```
em.getMetamodel().managedType(MyEntity.class);
```

Following the above code ObjectDB will include `MyEntity` in searching named queries.

4.1.4 JPA Criteria API Queries

The JPA Criteria API provides an alternative way for defining JPA queries, which is mainly useful for building dynamic queries whose exact structure is only known at runtime.

JPA Criteria API vs JPQL

JPQL queries are defined as strings, similarly to SQL. JPA criteria queries, on the other hand, are defined by instantiation of Java objects that represent query elements.

A major advantage of using the criteria API is that errors can be detected earlier, during compilation rather than at runtime. On the other hand, for many developers string based JPQL queries, which are very similar to SQL queries, are easier to use and understand.

For simple static queries - string based JPQL queries (e.g. as named queries) may be preferred. For dynamic queries that are built at runtime - the criteria API may be preferred.

For example, building a dynamic query based on fields that a user fills at runtime in a form that contains many optional fields - is expected to be cleaner when using the JPA criteria API, because it eliminates the need for building the query using many string concatenation operations.

String based JPQL queries and JPA criteria based queries are equivalent in power and in efficiency. Therefore, choosing one method over the other is also a matter of personal preference.

First JPA Criteria Query

The following query string represents a minimal JPQL query:

```
SELECT c FROM Country c
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
```

The `CriteriaBuilder` interface serves as the main factory of criteria queries and criteria query elements. It can be obtained either by the `EntityManagerFactory`'s `getCriteriaBuilder` method or by the `EntityManager`'s `getCriteriaBuilder` method (both methods are equivalent).

In the example above a `CriteriaQuery` instance is created for representing the built query.
Then a `Root` instance is created to define a range variable in the FROM clause. Finally, the range variable, `c`, is also used in the SELECT clause as the query result expression.

A `CriteriaQuery` instance is equivalent to a JPQL string and not to a `TypedQuery` instance. Therefore, running the query still requires a `TypedQuery` instance:

```
TypedQuery<Country> query = em.createQuery(q);
List<Country> results = query.getResultList();
```

Using the criteria API introduces some extra work, at least for simple static queries, since the equivalent JPQL query could simply be executed as follows:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Because eventually both types of queries are represented by a `TypedQuery` instance - query execution and query setting is similar, regardless of the way in which the query is built.

Parameters in Criteria Queries

The following query string represents a JPQL query with a parameter:

```
SELECT c FROM Country c WHERE c.population > :p
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
ParameterExpression<Integer> p = cb.parameter(Integer.class);
q.select(c).where(cb.gt(c.get("population"), p));
```

The `ParameterExpression` instance, `p`, is created to represent the query parameter. The where method sets the WHERE clause. As shown above, The `CriteriaQuery` interface supports method chaining. See the links in the next sections of this page for detailed explanations on how to set criteria query clauses and build criteria expressions.

Running this query requires setting the parameter:

```
TypedQuery<Country> query = em.createQuery(q);
query.setParameter(p, 10000000);
List<Country> results = query.getResultList();
```

The `serParameter` method takes a `Parameter` (or a `ParameterExpression`) instance as the first argument instead of a name or a position (which are used with string based JPQL parameters).

Criteria Query Structure

Queries in JPA (as in SQL) are composed of clauses. Because JPQL queries and criteria queries use equivalent clauses - they are explained side by side in the Query Structure pages.

Specific details about criteria query clauses are provided in the following page sections:

- SELECT clause (`select`, `distinct`, `multiselect`, `array`, `tuple`, `construct`).
- FROM clause (`from`, `join`, `fetch`).
- WHERE clause (`where`).
- GROUP BY / HAVING clauses (`groupBy`, `having`, `count`, `sum`, `avg`, `min`, `max`, ...).
- ORDER BY clause (`orderBy`, `Order`, `asc`, `desc`).

The links above are direct links to the criteria query sections in pages that describe query structure in general, including in the context of string based JPQL queries.

Criteria Query Expressions

JPA query clauses are composed of expressions. Because JPQL queries and criteria queries use equivalent expressions - they are explained side by side in the Query Expressions pages.

Specific details about criteria query expressions are provided in the following page sections:

- Literals and Dates (`literal`, `nullLiteral`, `currentDate`, ...).
- Paths, navigation and types (`get`, `type`).
- Arithmetic expressions (`sum`, `diff`, `prod`, `quot`, `mod`, `abs`, `neg`, `sqrt`).
- String expressions (`like`, `length`, `locate`, `lower`, `upper`, `concat`, `substring, ...`).
- Collection expressions (`isEmpty`, `isNotEmpty`, `isMember`, `isNotMember`, `size`).
- Comparison expressions (`equal`, `notEqual`, `gt`, `ge`, `lt`, `le`, `between`, `isNull`, ...)
- Logical expressions (`and`, `or`, `not`, `isTrue`).

The links above are direct links to the criteria query sections in pages that describe expressions in general, including in the context of string based JPQL queries.

4.1.5 Setting and Tuning of JPA Queries

The `Query` and `TypedQuery` interfaces define various setting and tuning methods that may affect query execution if invoked before a query is run using getResultList or getSingleResult.

Result Range (setFirstResult, setMaxResults)

The `setFirstResult` and `setMaxResults` methods enable defining a result window that exposes a portion of a large query result list (hiding anything outside that window). The `setFirstResult` method is used to specify where the result window begins, i.e. how many results at the beginning of the complete result list should be skipped and ignored. The `setMaxResults` method is used to specify the result window size. Any result after hitting that specified maximum is ignored.

These methods support the implementation of efficient result paging. For example, if each result page should show exactly `pageSize` results, and `pageId` represents the result page number (`0` for the first page), the following expression retrieves the results for a specified page:

```
List<Country> results =
    query.setFirstResult(pageIx * pageSize)
         .setMaxResults(pageSize)
         .getResultList();
```

These methods can be invoked in a single expression with `getResultList` since the setter methods in `Query` and `TypedQuery` support method chaining (by returning the query object on which they were invoked).

Flush Mode (setFlushMode)

Changes made to a database using an `EntityManager em` can be visible to anyone who uses `em`, even before committing the transaction (but not to users of other `EntityManager` instances). JPA implementations can easily make uncommitted changes visible in simple JPA operations, such as `find`. However, query execution is much more complex. Therefore, before a query is executed, uncommitted database changes (if any) have to be flushed to the database in order to be visible to the query.

Flush policy in JPA is represented by the `FlushModeType` enum, which has two values:

- `AUTO` - changes are flushed before query execution and on commit/flush.
- `COMMIT` - changes are flushed only on explicit commit/flush.

In most JPA implementations the default is `AUTO`. In ObjectDB the default is `COMMIT` (which is more efficient).

The default mode can be changed by the application, either at the `EntityManager` level as a default for all the queries in that `EntityManager` or at the level of a specific query, by overriding the default `EntityManager` setting:

```
// Enable query time flush at the EntityManager level:
em.setFlushMode(FlushModeType.AUTO);

// Enable query time flush at the level of a specific query:
query.setFlushMode(FlushModeType.AUTO);
```

Flushing changes to the database before every query execution affects performance significantly. Therefore, when performance is important, this issue has to be considered.

Lock Mode (setLockMode)

ObjectDB uses automatic optimistic locking to prevent concurrent changes to entity objects by multiple users. JPA 2 adds support for pessimistic locking. The `setLockMode` method sets a lock mode that has to be applied on all the result objects that the query retrieves. For example, the following query execution sets a pessimistic WRITE lock on all the result objects:

```
List<Country> results =
    query.setLockMode(LockModeType.PESSIMISTIC_WRITE)
         .getResultList();
```

Notice that when a query is executed with a requested pessimistic lock mode it could fail if locking fails, throwing a `LockTimeoutException`.

Query Hints (Execution Timeout and Lock Timeout)

Additional settings can be applied to queries via hints.

Supported Query Hints

ObjectDB supports the following query hints:

- `"javax.persistence.query.timeout"` - sets maximum query execution time in milliseconds. A `QueryTimeoutException` is thrown if timeout is exceeded.
- `"javax.persistence.lock.timeout"` - sets maximum waiting time for pessimistic locks, when pessimistic locking of query results is enabled. See the Lock Timeout section for more details about lock

timeout.

- `"objectdb.query-language"` - sets the query language, as one of `"JPQL"` (JPA query language), `"JDOQL"` (JDO query language) or `"ODBQL"` (ObjectDB query language). The default is ODBQL, which is a union of JPQL, JDOQL and ObjectDB extensions. Setting `"JPQL"` is useful to enforce portable JPA code by ObjectDB.
- `"objectdb.result-fetch"` - sets fetch mode for query result as either `"EAGER"` (the default) or `"LAZY"`. When LAZY is used result entity objects are returned as references (with no content). This could be useful when the shared L2 cache is enabled and entity objects may already be available in the cache.

Setting Query Hint (Scopes)

Query hints can be set in the following scopes (from global to local):

For the entire persistence unit - using a `persistence.xml` property:

```
<properties>
   <property name="javax.persistence.query.timeout" value="3000"/>
</properties>
```

For an `EntityManagerFactory` - using the `createEntityManagerFacotory` method:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.query.timeout", 4000);
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("pu", properties);
```

For an `EntityManager` - using the `createEntityManager` method:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.query.timeout", 5000);
EntityManager em = emf.createEntityManager(properties);
```

or using the `setProperty` method:

```
em.setProperty("javax.persistence.query.timeout", 6000);
```

For a named query definition - using the `hints` element:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c",
    hints={@QueryHint(name="javax.persistence.query.timeout", value="7000")})
```

For a specific query execution - using the `setHint` method (before query execution):

```
  query.setHint("javax.persistence.query.timeout", 8000);
```

A hint that is set in a global scope affects all the queries in that scope (unless it is overridden in a more local scope). For example, setting a query hint in an `EntityManager` affects all the queries that are created in that `EntityManager` (except queries with explicit setting of the same hint).

4.2 JPA Query Structure (JPQL / Criteria)

The syntax of the Java Persistence Query Language (JPQL) is very similar to the syntax of SQL. Having a SQL like syntax in JPA queries is an important advantage because SQL is a very powerful query language and many developers are already familiar with it.

The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, whereas JPQL works with Java classes and objects. For example, a JPQL query can retrieve and return entity objects rather than just field values from database tables, as with SQL. That makes JPQL more object oriented friendly and easier to use in Java.

JPQL Query Structure

As with SQL, a JPQL SELECT query also consists of up to 6 clauses in the following format:

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

The first two clauses, SELECT and FROM are required in every retrieval query (update and delete queries have a slightly different form). The other JPQL clauses, WHERE, GROUP BY, HAVING and ORDER BY are optional.

The structure of JPQL DELETE and UPDATE queries is simpler:

```
DELETE FROM ... [WHERE ...]

UPDATE ... SET ... [WHERE ...]
```

Besides a few exceptions, JPQL is case insensitive. JPQL keywords, for example, can appear in queries either in upper case (e.g. SELECT) or in lower case (e.g. select). The few exceptions in which JPQL is case sensitive include mainly Java source elements such as names of entity classes and persistent fields, which are case sensitive. In addition, string literals are also case sensitive (e.g. "ORM" and "orm" are different values).

A Minimal JPQL Query

The following query retrieves all the `Country` objects in the database:

```
SELECT c FROM Country AS c
```

Because SELECT and FROM are mandatory, this demonstrates a minimal JPQL query.

The FROM clause declares one or more query variables (also known as identification variables). Query variables are similar to loop variables in programing languages. Each query variable represents iteration over objects in the database. A query variable that is bound to an entity class is referred to as a range variable. Range variables define iteration over all the database objects of a binding entity class and its descendant classes. In the query above, `c` is a range variable that is bound to the `Country` entity class and defines iteration over all the `Country` objects in the database.

The SELECT clause defines the query results. The query above simply returns all the `Country` objects from the iteration of the `c` range variable, which in this case is actually all the `Country` objects in the database.

Organization of this Section

This section contains the following pages:

- SELECT clause (JPQL / Criteria API)
- FROM clause (JPQL / Criteria API)
- WHERE clause (JPQL / Criteria API)
- GROUP BY and HAVING clauses
- ORDER BY clause (JPQL / Criteria API)

- DELETE Queries in JPA/JPQL
- UPDATE Queries in JPA/JPQL

Detailed explanations on how to set criteria query clauses are provided as follows:

- Criteria SELECT (`select`, `distinct`, `multiselect`, `array`, `tuple`, `construct`).
- Criteria FROM (`from`, `join`, `fetch`).
- Criteria WHERE (`where`).
- Criteria GROUP BY / HAVING (`groupBy`, `having`, `count`, `sum`, `avg`, `min`, `max`, ...).
- Criteria ORDER BY (`orderBy`, `Order`, `asc`, `desc`).

4.2.1 SELECT clause (JPQL / Criteria API)

The ability to retrieve managed entity objects is a major advantage of JPQL. For example, the following query returns `Country` objects that become managed by the `EntityManager em`:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Because the results are managed entity objects they have all the support that JPA provides for managed entity objects, including transparent navigation to other database objects, transparent update detection, support for delete, etc.

Query results are not limited to entity objects. JPA 2 adds the ability to use almost any valid JPQL expression in SELECT clauses. Specifying the required query results more precisely can improve performance and in some cases can also reduce the amount of Java code needed. Notice that query results must always be specified explicitly - JPQL does not support the "SELECT *" expression (which is commonly used in SQL).

Projection of Path Expressions

JPQL queries can also return results which are not entity objects. For example, the following query returns country names as `String` instances, rather than `Country` objects:

```
SELECT c.name FROM Country AS c
```

Using path expressions, such as `c.name`, in query results is referred to as projection. The field values are

extracted from (or projected out of) entity objects to form the query results.

The results of the above query are received as a list of `String` values:

```
TypedQuery<String> query = em.createQuery(
    "SELECT c.name FROM Country AS c", String.class);
List<String> results = query.getResultList();
```

Only singular value path expressions can be used in the SELECT clause. Collection and map fields cannot be included in the results directly, but their content can be added to the SELECT clause by using a bound JOIN variable in the FROM clause.

Nested path expressions are also supported. For example, the following query retrieves the name of the capital city of a specified country:

```
SELECT c.capital.name FROM Country AS c WHERE c.name = :name
```

Because construction of managed entity objects has some overhead, queries that return non entity objects, as the two queries above, are usually more efficient. Such queries are useful mainly for displaying information efficiently. They are less productive with operations that update or delete entity objects, in which managed entity objects are needed.

Managed entity objects can, however, be returned from a query that uses projection when a result path expression resolves to an entity. For example, the following query returns a managed `City` entity object:

```
SELECT c.capital FROM Country AS c WHERE c.name = :name
```

Result expressions that represent anything but entity objects (e.g. values of system types and user defined embeddable objects) return as results value copies that are not associated with the containing entities. Therefore, embedded objects that are retrieved directly by a result path expression are not associated with an EntityManager and changes to them when a transaction is active are not propagated to the database.

Multiple SELECT Expressions

The SELECT clause may also define composite results:

```
SELECT c.name, c.capital.name FROM Country AS c
```

The result list of this query contains `Object[]` elements, one per result. The length of each result `Object[]` element is 2. The first array cell contains the country name (`c.name`) and the second array cell contains the capital city name (`c.capital.name`).

The following code demonstrates running this query and processing the results:

```
TypedQuery<Object[]> query = em.createQuery(
    "SELECT c.name, c.capital.name FROM Country AS c", Object[].class);
List<Object[]> results = query.getResultList();
for (Object[] result : results) {
    System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

As an alternative to representing compound results by `Object` arrays, JPA supports using custom result classes and result constructor expressions.

Result Classes (Constructor Expressions)

JPA supports wrapping JPQL query results with instances of custom result classes. This is mainly useful for queries with multiple SELECT expressions, where custom result objects can provide an object oriented alternative to representing results as `Object[]` elements.

The fully qualified name of the result class is specified in a NEW expression, as follows:

```
SELECT NEW example.CountryAndCapital(c.name, c.capital.name)
FROM Country AS c
```

This query is identical to the previous query above except that now the result list contains `CountryAndCapital` instances rather than `Object[]` elements.

The result class must have a compatible constructor that matches the SELECT result expressions, as follows:

```
package example;

public class CountryAndCapital {
    public String countryName;
    public String capitalName;

    public CountryAndCapital(String countryName, String capitalName) {
        this.countryName = countryName;
        this.capitalName = capitalName;
```

```
    }
}
```

The following code demonstrates running this query:

```
  String queryStr =
      "SELECT NEW example.CountryAndCapital(c.name, c.capital.name) " +
      "FROM Country AS c";
  TypedQuery<CountryAndCaiptal> query =
      em.createQuery(queryStr, CountryAndCaiptal.class);
  List<CountryAndCaiptal> results = query.getResultList();
```

Any class with a compatible constructor can be used as a result class. It could be a JPA managed class (e.g. an entity class) but it could also be a lightweight 'transfer' class that is only used for collecting and processing query results.

If an entity class is used as a result class, the result entity objects are created in the NEW state, which means that they are not managed. Such entity objects are missing the JPA functionality of managed entity objects (e.g. transparent navigation and transparent update detection), but they are more lightweight, they are built faster and they consume less memory.

SELECT DISTINCT

Queries that use projection may return duplicate results. For example, the following query may return the same currency more than once:

```
SELECT c.currency FROM Country AS c WHERE c.name LIKE &#39;I%&#39;
```

Both Italy and Ireland (whose name starts with 'I') use Euro as their currency. Therefore, the query result list contains `"Euro"` more than once.

Duplicate results can be eliminated easily in JPQL by using the DISTINCT keyword:

```
SELECT DISTINCT c.currency FROM Country AS c WHERE c.name LIKE &#39;I%&#39;
```

The only difference between SELECT and SELECT DISTINCT is that the later filters duplicate results. Filtering duplicate results might have some effect on performance, depending on the size of the query result list and other factors.

SELECT in Criteria Queries

The criteria query API provides several ways for setting the SELECT clause.

Single Selection

Setting a single expression SELECT clause is straightforward.

For example, the following JPQL query:

```
SELECT DISTINCT c.currency FROM Country c
```

can be built as a criteria query as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c.get("currency")).distinct(true);
```

The `select` method takes one argument of type `Selection` and sets it as the SELECT clause content (overriding previously set SELECT content if any). Every valid criteria API expression can be used as selection, because all the criteria API expressions are represented by a sub interface of `Selection` - `Expression` (and its descendant interfaces).

The distinct method can be used to eliminate duplicate results as demonstrated in the above code (using method chaining).

Multi Selection

The `Selection` interface is also a super interface of `CompoundSelection`, which represents multi selection (which is not a valid expression by its own and can be used only in the SELECT clause).

The `CriteriaBuilder` interface provides three factory methods for building `CompoundSelection` instances - `array`, `tuple` and `construct`.

CriteriaBuilder's array

The following JPQL query:

```
SELECT c.name, c.capital.name FROM Country c
```

can be defined using the criteria API as follows:

```
CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
Root<Country> c = q.from(Country.class);
q.select(cb.array(c.get("name"), c.get("capital").get("name")));
```

The `array` method builds a `CompoundSelection` instance, which represents results as arrays.

The following code demonstrates execution of the query and iteration over the results:

```
List<Object[]> results = em.createQuery(q).getResultList();
for (Object[] result : results) {
    System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

CriteriaBuilder's tuple

The `Tuple` interface can be used as a clean alternative to `Object[]`:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Country> c = q.from(Country.class);
q.select(cb.tuple(c.get("name"), c.get("capital").get("name")));
```

The `tuple` method builds a `CompoundSelection` instance, which represents `Tuple` results.

The following code demonstrates execution of the query and iteration over the results:

```
List<Tuple> results = em.createQuery(q).getResultList();
for (Tuple t : results) {
    System.out.println("Country: " + t.get(0)  + ", Capital: " + t.get(1));
}
```

The `Tuple` interface defines several other methods for accessing the result data.

CriteriaBuilder's construct

JPQL user defined result objects are also supported by the JPA criteria query API:

```
CriteriaQuery<CountryAndCapital> q = cb.createQuery(CountryAndCapital.class);
Root<Country> c = q.from(Country.class);
q.select(cb.construct(CountryAndCapital.class,
    c.get("name"), c.get("capital").get("name")));
```

The `construct` method builds a `CompoundSelection` instance, which represents results as instances of a user defined class (`CountryAndCapital` in the above example).

The following code demonstrates execution of the query:

```
List<CountryAndCapital> results = em.createQuery(q).getResultList();
```

As expected - the result objects are `CountryAndCapital` instances.

CriteriaQuery's multiselect

In the above examples, `CompoundSelection` instances were first built by a `CriteriaBuilder` factory method and then passed to the `CriteriaQuery`'s `select` method.

The `CriteriaQuery` interface provides a shortcut method - `multiselect`, which takes a variable number of arguments representing multiple selections, and builds a `CompoundSelection` instance based on the expected query results.

For example, the following invocation of `multiselect`:

```
q.multiselect(c.get("name"), c.get("capital").get("name"));
```

is equivalent to using `select` with one of the factory methods (`array`, `tuple` or `construct`) as demonstrated above.

The behavior of the `multiselect` method depends on the query result type (as set when `CriteriaQuery` is instantiated):

- For expected `Object` and `Object[]` result type - `array` is used.
- For expected `Tuple` result - `tuple` is used.
- For any other expected result type - `construct` is used.

4.2.2 FROM clause (JPQL / Criteria API)

The FROM clause declares query identification variables that represent iteration over objects in the database. A query identification variable is similar to a variable of a Java enhanced for loop in a program, since both are used for iteration over objects.

Range Variables

Range variables are query identification variables that iterate over all the database objects of a specific entity class hierarchy (i.e. an entity class and all its descendant entity classes). Identification variables are always polymorphic. JPQL does not provide a way to exclude descendant classes from iteration at the FROM clause level. JPA 2, however, adds support for filtering instances of specific types at the WHERE clause level by using a type expression.

For example, in the following query, c iterates over all the `Country` objects in the database:

```
SELECT c FROM Country AS c
```

The AS keyword is optional, and the same query can also be written as follows:

```
SELECT c FROM Country c
```

By default, the name of an entity class in a JPQL query is the unqualified name of the class (e.g. just `Country` with no package name). The default name can be overridden by specifying another name explicitly in the `@Entity's name` annotation element.

Multiple range variables are allowed. For example, the following query returns all the pairs of countries that share a common border:

```
SELECT c1, c2 FROM Country c1, Country c2
WHERE c2 MEMBER OF c1.neighbors
```

Multiple variables are equivalent to nested loops in a program. The FROM clause above defines two loops. The outer loop uses `c1` to iterate over all the `Country` objects. The inner loop uses `c2` to also iterate over all the `Country` objects. A similar query with no `WHERE` clause would return all the possible combinations of two countries. The `WHERE` clause filters any pair of countries that do not share a border, returning as results only neighbor countries.

Caution is required when using multiple range variables. Iteration over about 1,000,000 database objects with a single range variable might be acceptable. But iteration over the same objects with two range variables forming nested loops (outer and inner) might prevent query execution within a reasonable response time.

Database Management Systems (DBMS), including ObjectDB, try to optimize execution of multi-variable queries. Whenever possible, full nested iteration over the entire Cartesian product is avoided. The above query, for example, can be executed as follows. An outer loop iterates with `c1` over all the `Country` objects in the database. An inner loop iterates with `c2` only over the `neighbors` collection of the outer `c1`. In this case, by propagation of a WHERE constraint to the FROM phase, a full iteration over the Cartesian product is avoided.

[INNER] JOIN

As discussed above, range variables represent iteration over all the database objects of a specified entity type. JPQL provides an additional type of identification variable, a join variable, which represent a more limited iteration over specified collections of objects.

The following query uses one range variable and one join variable:

```
SELECT c1, c2 FROM Country c1 INNER JOIN c1.neighbors c2
```

In JPQL, JOIN can only appear in a FROM clause. The INNER keyword is optional (i.e. INNER JOIN is equivalent to JOIN). `c1` is declared as a range variable that iterates over all the `Country` objects in the database. `c2` is declared as a join variable that is bound to the `c1.neighbors` path and iterates only over objects in that collection.

You might have noticed that this query is equivalent to the previous neighbors query, which has two range variables and a WHERE clause. However, this second query form that uses a join variable is preferred. Besides being shorter and cleaner, the second query describes the right and efficient way for executing the query (which is using a full range outer loop and a collection limited inner loop) without relying on DBMS optimizations.

It is quite common for JPQL queries to have a single range variable that serves as a root and additional join variables that are bound to path expressions. Join variables can also be bound to path expressions that are based on other join variables that appear earlier in the FROM clause.

Join variables can also be bound to a single value path expression. For example:

```

```

```
SELECT c, p.name FROM Country c JOIN c.capital p
```

In this case, the inner loop iterates over a single object because every `Country c` has only one `Capital p`. Join variables that are bound to a single value expression are less commonly used because usually they can be replaced by a simpler long path expression (which is not an option for a collection). For example:

```
SELECT c, c.capital.name FROM Country c
```

One exception is when OUTER JOIN is required because path expressions function as implicit INNER JOIN variables.

LEFT [OUTER] JOIN

To understand the purpose of OUTER JOIN, consider the following INNER JOIN query that retrieves pairs of (country name, capital name):

```
SELECT c.name, p.name FROM Country c JOIN c.capital p
```

The FROM clause defines iteration over (country, capital) pairs. A country with no capital city (e.g. Nauru, which does not have an official capital) is not part of any iterated pair and is therefore excluded from the query results. INNER JOIN simply skips any outer variable value (e.g. any `Country`) that has no matching inner variable (e.g. a `Capital`).

The behavior of OUTER JOIN is different, as demonstrated by the following query variant:

```
SELECT c, p.name FROM Country c LEFT OUTER JOIN c.capital p
```

The OUTER keyword is optional (LEFT OUTER JOIN is equivalent to LEFT JOIN). When using OUTER JOIN, if a specific outer variable does not have any matching inner value it gets at least a NULL value as a matching value in the FROM iteration. Therefore, a `Country c` with no `Capital` city has a minimum representation of (c, NULL) in the FROM iteration.

For example, unlike the INNER JOIN variant of this query that skips Nauru completely, the OUTER JOIN variant returns Nauru with a NULL value as its capital.

[LEFT [OUTER] | INNER] JOIN FETCH

JPA support of transparent navigation and fetch makes it very easy to use, since it provides the illusion that all the database objects are available in memory for navigation. But this feature could also cause performance problems.

For example, let's look at the following query execution and result iteration:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
for (Country c : results) {
    System.out.println(c.getName() + " => " + c.getCapital().getName());
}
```

The query returns only `Country` instances. Consequently, the loop that iterates over the results is inefficient, since retrieval of the referenced `Capital` objects is performed one at a time, i.e. the number of round trips to the database is much larger than necessary.

A simple solution is to use the following query, which returnes exactly the same result objects (`Country` instances):

```
SELECT c FROM Country c JOIN FETCH c.capital
```

The JOIN FETCH expression is not a regular JOIN and it does not define a JOIN variable. Its only purpose is specifying related objects that should be fetched from the database with the query results on the same round trip. Using this query improves the efficiency of iteration over the result `Country` objects because it eliminates the need for retrieving the associated `Capital` objects separately.

Notice, that if the Coutry and Capital objects are needed only for their names - the following report query could be even more efficient:

```
SELECT c.name, c.capital.name FROM Country c
```

Reserved Identifiers

The name of a JPQL query variable must a valid Java identifier but cannot be one of the following reserved words:

`ABS, ALL, AND, ANY, AS, ASC, AVG, BETWEEN, BIT_LENGTH, BOTH, BY, CASE, CHAR_LENGTH, CHARACTER_LENGTH, CLASS, COALESCE, CONCAT, COUNT, CURRENT_DATE, CURRENT_TIME,`

CURRENT_TIMESTAMP,DELETE, DESC, DISTINCT, ELSE, EMPTY, END, ENTRY, ESCAPE, EXISTS, FALSE, FETCH,FROM, GROUP, HAVING, IN, INDEX, INNER, IS, JOIN, KEY, LEADING, LEFT, LENGTH, LIKE,LOCATE, LOWER, MAX, MEMBER, MIN, MOD, NEW, NOT, NULL, NULLIF, OBJECT, OF, OR, ORDER, OUTER, POSITION, SELECT, SET, SIZE, SOME, SQRT, SUBSTRING, SUM, THEN,TRAILING, TRIM, TRUE, TYPE, UNKNOWN, UPDATE, UPPER, VALUE, WHEN, WHERE.

JPQL variables as well as all the reserved identifiers in the list above are case insensitive. Therefore, ABS, abs, Abs and aBs are all invalid variable names.

FROM and JOIN in Criteria Queries

FROM query identification variables are represented in criteria queries by sub interfaces of `From`:

- Range variables are represented by the `Root` interface.
- Join variables are represented by the `Join` interface (and its sub interfaces).

Criteria Query Roots

The `CriteriaQuery`'s `from` method serves as a factory of `Root` instances.

For example, the following JPQL query, which defines two uncorrelated range variables - `c1`, `c2`:

```
SELECT c1, c2 FROM Country c1, Country c2
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c1 = q.from(Country.class);
Root<Country> c2 = q.from(Country.class);
q.multiselect(c1, c2);
```

Unlike other `CriteriaQuery` methods - invocation of the `from` method does not override a previous invocation of that method. Every time the `from` method is invoked - a new variable is added to the query.

Criteria Query Joins

JOIN variables are represented in criteria queries by the `Join` interface (and its sub interfaces).

For example, the following JPQL query:

```
SELECT c, p.name FROM Country c LEFT OUTER JOIN c.capital p
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
Join<Country> p = c.join("capital", JoinType.LEFT);
q.multiselect(c, p.get("name"));
```

The `From` interface provides various forms of the `join` method. Every invocation of `join` adds a new JOIN variable to the query. Since `From` is the super interface of both `Root` and `Join` - `join` methods can be invoked on `Root` instances (as demonstrated above) as well as on previously built `Join` instances.

Criteria Query Fetch Joins

The following JPQL query:

```
SELECT c FROM Country c JOIN FETCH c.capital
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
Fetch<Country,Capital> p = c.fetch("capital");
q.select(c);
```

Several forms of the `fetch` method are defined in the `Path` interface, which represents path expressions and is also the super interface of the `Root` and `Join` interfaces.

4.2.3 WHERE clause (JPQL / Criteria API)

The WHERE clause adds filtering capabilities to the FROM-SELECT structure. It is essential in any JPQL query that retrieves selective objects from the database. Out of the four optional clauses of JPQL queries, the WHERE clause is definitely the most frequently used.

How a WHERE Clause Works

The following query retrieves only countries with population size that exceeds a specified limit, which is represented by the parameter p:

```
SELECT c FROM Country c WHERE c.population > :p
```

The FROM clause of this query defines an iteration over all the `Country` objects in the database using the `c` range variable. Before passing these `Country` objects to the SELECT clause for collecting as query results, the WHERE clause gets an opportunity to function as a filter. The boolean expression in the WHERE clause, which is also known as the WHERE predicate, defines which objects to accept. Only `Country` objects for which the predicate expression evaluates to TRUE are passed to the SELECT clause and then collected as query results.

WHERE Predicate and Indexes

Formally, the WHERE clause functions as a filter between the FROM and the SELECT clauses. Practically, if a proper index is available, filtering is done earlier during FROM iteration. In the above population query, if an index is defined on the `population` field ObjectDB can use that index to iterate directly on `Country` objects that satisfy the WHERE predicate. For entity classes with millions of objects in the database there is a huge difference in query execution time if proper indexes are defined.

WHERE Filter in Multi Variable Queries

In a multi-variable query the FROM clause defines iteration on tuples. In this case the WHERE clause filters tuples before passing them to the SELECT clause.

For example, the following query retrieves all the countries with population size that exceeds a specified limit and also have an official language from a specified set of languages:

```
SELECT c, l FROM Country c JOIN c.languages l
WHERE c.population > :p AND l in :languages
```

The FROM clause of this query defines iteration over (country, language) pairs. Only pairs that satisfy the WHERE clause are passed through to the SELECT.

In multi-variable queries the number of tuples for iteration might be very large even if the database is small, making indexes even more essential.

JPQL Expressions in WHERE

The above queries demonstrate only a small part of the full capabilities of a WHERE clause.
The real power of the JPQL WHERE clause is derived from the rich JPQL expression syntax,
which includes many operators (arithmetic operators, relational operators, logical operators) and functions (numeric functions, string functions, collection functions). The WHERE predicate is always a boolean JPQL expression. JPQL expressions are also used in other JPQL query clauses but they are especially dominant in the WHERE clause.

WHERE in Criteria Queries

The `CriteriaQuery` interface provides two `where` methods for setting the WHERE clause.

Single Restriction

The first `where` method takes one `Expression<Boolean>` argument and uses it as the WHERE clause content (overriding previously set WHERE content if any).

For example, the following JPQL query:

```
SELECT c FROM Country c WHERE c.population > :p
```

can be built by using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
ParameterExpression<Integer> p = cb.parameter(Integer.class);
q.where(cb.gt(c.get("population"), p));
```

Multiple Restrictions

The second `where` method takes a variable number of arguments of `Predicate` type and uses an AND conjunction as the WHERE clause content (overriding previously set WHERE content if any):

For example, the following JPQL query:

```
SELECT c FROM Country WHERE c.population > :p AND c.area < :a
```

can be built as a criteria query as follows:

```
CriteriaQuery q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
ParameterExpression<Integer> p = cb.parameter(Integer.class);
ParameterExpression<Integer> a = cb.parameter(Integer.class);
q.where(
    cb.gt(c.get("population"), p),
    cb.lt(c.get("area"), a)
);
```

The `where` setting above is equivalent to explicitly building an AND conjunction, as so:

```
q.where(
    cb.and(
        cb.gt(c.get("population"), p),
        cb.lt(c.get("area"), a)
    )
);
```

The variable argument form of the `where` method always uses AND. Therefore, using OR requires building an OR expression explicitly:

```
q.where(
    cb.or(
        cb.gt(c.get("population"), p),
        cb.lt(c.get("area"), a)
    )
);
```

See the Logical Operators page for explanations on boolean expressions and predicates that can be used in a criteria query WHERE clause.

4.2.4 GROUP BY and HAVING clauses

The GROUP BY clause enables grouping of query results. A JPQL query with a GROUP BY clause returns properties of generated groups instead of individual objects and fields.

The position of a GROUP BY clause in the query execution order is after the FROM and WHERE clauses, but before the SELECT clause. When a GROUP BY clause exists in a JPQL query, database objects (or tuples of database objects) that are generated by the FROM clause iteration and pass the WHERE clause filtering (if any) are sent to grouping by the GROUP BY clauses before arriving at the SELECT clause.

GROUP BY as DISTINCT (no Aggregates)

The following query groups all the countries by their first letter:

```
SELECT SUBSTRING(c.name, 1, 1)
FROM Country c
GROUP BY SUBSTRING(c.name, 1, 1);
```

The FROM clause defines iteration over all the `Country` objects in the database. The GROUP BY clause groups these `Country` objects by the first letter of the country name. The next step is to pass the groups to the SELECT clause which returns the first letters as result.

ObjectDB is very flexible in allowing JPQL expressions anywhere in the query. Note that the query above might not be supported by some JPA implementations. Only identification variables and path expressions are currently supported in the GROUP BY clause by all the JPA implementations.

Grouping the `Country` objects makes them inaccessible to the SELECT clause as individuals. Therefore, the SELECT clause can only use properties of the groups, which include:

- The properties that are used for grouping (each group has unique value combination).
- Aggregate calculations (count, sum, avg, max, min) that are carried out on all the objects (or the object tuples) in the group.

The aggregate calculation gives the GROUP BY clause its power. Actually, without aggregate calculations - the GROUP BY functions merely as a DISTINCT operator. For example, the above query (which does not use aggregates) is equivalent to the following query:

```
SELECT DISTINCT SUBSTRING(c.name, 1, 1) FROM Country c
```

GROUP BY with Aggregate Functions

JPQL supports the five aggregate functions of SQL:

- `COUNT` - returns a `long` value representing the number of elements.

- SUM - returns the sum of numeric values.
- AVG - returns the average of numeric values as a double value.
- MIN - returns the minimum of comparable values (numeric, strings, dates).
- MAX - returns the maximum of comparable values (numeric, strings, dates).

The following query counts for every letter the number of countries with names that start with that letter and the number of different currencies that are used by these countries:

```
SELECT SUBSTRING(c.name, 1, 1), COUNT(c), COUNT(DISTINCT c.currency)
FROM Country c
GROUP BY SUBSTRING(c.name, 1, 1);
```

The query returns Object[] arrays of length 3, in which the first cell contains the initial letter as a String object, the second cell contains the number of countries in that letter's group as a Long object and the third cell contains the distinct number of currencies that are in use by countries in that group. The DISTINCT keyword in a COUNT aggregate expression, as demonstrated above), eliminates duplicate values when counting.

Only the COUNT aggregate function can be applied to entity objects directly. Other aggregate functions are applied to fields of objects in the group by using path expressions.

The following query groups countries in Europe by their currency, and for each group returns the currency and the cumulative population size in countries that use that currency:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
```

Because grouping is performed in this query on a path expression, this query is standard and it is expected to be supported by all JPA implementations.

GROUP BY with HAVING

Groups in JPQL grouping queries can be filtered using the HAVING clause. The HAVING clause for the GROUP BY clause is like the WHERE clause for the FROM clause. ObjectDB supports the HAVING clause only when a GROUP BY clause exists.

The following query uses HAVING to change the previous query in a way that single country groups are ignored:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
HAVING COUNT(c) > 1
```

The HAVING clause stands as a filter between the GROUP BY clause and the SELECT clause in such a way that only groups that are accepted by the HAVING filter are passed to the SELECT clause. The same restrictions on SELECT clause in grouping queries also apply to the HAVING clause, which means that individual object fields are inaccessible. Only group properties which are the expressions that are used for grouping (e.g. c.currency) and aggregate expressions are allowed in the HAVING clause.

Global Aggregates (no GROUP BY)

JPQL supports a special form of aggregate queries that do not have a GROUP BY clause in which all the FROM/WHERE objects (or object tuples) are considered as one group.

For example, the following query returns the sum and average population in countries that use the English language:

```
SELECT SUM(c.population), AVG(c.population)
FROM Country c
WHERE 'English' MEMBER OF c.languages
```

All the Country objects that pass the FROM/WHERE phase are considered as one group, for which the cumulative population size and the average population size is then calculated. Any JPQL query that contains an aggregate expression in the SELECT clause is considered a grouping query with all the attached restrictions, even when a GROUP BY clause is not specified. Therefore, in this case, only aggregate functions can be specified in the SELECT clause and individual objects and their fields become inaccessible.

GROUP BY and HAVING in Criteria Queries

The `CriteriaQuery` interface provides methods for setting the GROUP BY and HAVING clauses.

For example, the following JPQL query:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
```

```
GROUP BY c.currency
HAVING COUNT(c) > 1
```

can be built using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c.get("currency"), cb.sum(c.get("population")));
q.where(cb.isMember("Europe", c.get("continents")));
q.groupBy(c.get("currency"));
g.having(cb.gt(cb.count(c), 1));
```

The `CriteriaBuilder` interface provides methods for building aggregate expressions:

- `count`, `countDistinct` - return a long type expression representing the number of elements.
- `sum` , `sumAsLong`, `sumAsDouble` - return an expression representing the sum of values.
- `avg` - returns a double type expression representing the average of numeric values.
- `min`, `least` - return an expression representing the minimum of comparable values.
- `max`, `greatest` - return an expression representing the maximum of comparable values.

The `groupBy` method takes a variable number of arguments specifying one or more grouping expressions (or a list of expressions in another form of `groupBy`).

Setting a HAVING clause is very similar to setting a WHERE clause. As with the WHERE clause - two forms of the `having` method are provided. One `having` form takes an `Expression<Boolean>` argument and the other `having` form takes a variable number of `Predicate` arguments (and uses an AND conjunction).

When a `groupBy` or a `having` method is invoked, previously set values (if any) are discarded.

4.2.5 ORDER BY clause (JPQL / Criteria API)

The ORDER BY clause specifies a required order for the query results. Any JPQL query that does not include an ORDER BY clause produces results in an undefined and non-deterministic order.

ORDER BY Expressions

The following query returns names of countries whose population size is at least one million people, ordered by the country name:

```
SELECT c.name FROM Country c WHERE c.population > 1000000 ORDER BY c.name
```

When an ORDER BY clause exists it is the last to be executed. First the FROM clause produces objects for examination and the WHERE clause selects which objects to collect as results. Then the SELECT clause builds the results by evaluating the result expressions. Finally the results are ordered by evaluation of the the ORDER BY expressions.

Only expressions that are derived from expressions in the SELECT clause are allowed in the ORDER BY clause. The following query, for example, is invalid because the ORDER BY expression is not part of the results:

```
SELECT c.name
FROM Country c
WHERE c.population > 1000000
ORDER BY c.population
```

On the other hand, the following query is valid because, given a `Country c`, the `c.population` expression can be evaluated from `c`:

```
SELECT c
FROM Country c
WHERE c.population > 1000000
ORDER BY c.population
```

When using ObjectDB, any JPQL expression whose type is comparable (i.e. numbers, strings and date values) and is derived from the SELECT expressions can be used in the ORDER BY clause. Some JPA implementation are more restrictive. Path expressions are supported by all the JPA implementations but support of other JPQL expressions is vendor dependent.

Query results can also be ordered by multiple order expressions. In this case, the first expression is the primary order expression. Any additional order expression is used to order results for which all the previous order expressions produce the same values.

The following query returns `Country` objects ordered by `currency` as the primary sort key and by `name` as the secondary sort key:

```
SELECT c.currency, c.name
FROM Country c
```

```
ORDER BY c.currency, c.name
```

To avoid repeating result expressions in the ORDER BY JPQL supports defining aliases for SELECT expressions and then using the aliases in the ORDER BY clause. The following query is equivalent to the query above:

```
SELECT c.currency AS currency, c.name AS name
FROM Country c
ORDER BY currency, name
```

Alias variables are referred to as result variables to distinguish them from the identification variables that are defined in the FROM clause.

Order Direction (ASC, DESC)

The default ordering direction is ascending. Therefore, when ascending order is required it is usually omitted even though it could be specified explicitly, as follows:

```
SELECT c.name FROM Country c ORDER BY c.name ASC
```

On the other hand, to apply descending order the DESC keyword must be added explicitly to the order expression:

```
SELECT c.name FROM Country c ORDER BY c.name DESC
```

Grouping (GROUP BY) Order

The ORDER BY clause is always the last in the query processing chain. If a query contains both an ORDER BY clause and a GROUP BY clause the SELECT clause receives groups rather than individual objects and ORDER BY can order these groups. For example:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
HAVING COUNT(c) > 1
```

```
ORDER BY c.currency
```

The ORDER BY clause in the above query orders the results by the currency name. Without an ORDER BY clause the result order would be undefined.

ORDER BY in Criteria Queries

The `CriteriaQuery` interface provides methods for setting the ORDER BY clause.

For example, the following JPQL query:

```
SELECT c
FROM Country c
ORDER BY c.currency, c.population DESC
```

can be built using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
q.orderBy(cb.asc(c.get("currency")), cb.desc(c.get("population")));
```

Unlike other methods for setting criteria query clauses - the `orderBy` method takes a variable number of `Order` instances as arguments (or a list of `Order`) rather than `Expression` instances.

The `Order` interface is merely a thin wrapper around `Expression`, which adds order direction - either ascending (ASC) or descending (DESC). The `CriteriaBuilder`'s `asc` and `desc` methods (which are demonstrated above) take an expression and return an ascending or descending `Order` instance (respectively).

4.2.6 DELETE Queries in JPA/JPQL

As explained in chapter 2, entity objects can be deleted from the database by:

- Retrieving the entity objects into an `EntityManager`.
- Removing these objects from the `EntityManager` within an active transaction, either explicitly by calling the `remove` method or implicitly by a cascading operation.
- Applying changes to the database by calling the `commit` method.

JPQL DELETE queries provide an alternative way for deleting entity objects. Unlike SELECT queries, which are used to retrieve data from the database, DELETE queries do not retrieve data from the database, but when executed, delete specified entity objects from the database.

Removing entity objects from the database using a DELETE query may be slightly more efficient than retrieving entity objects and then removing them, but it should be used cautiously because bypassing the `EntityManager` may break its synchronization with the database. For example, the `EntityManager` may not be aware that a cached entity object in its persistence context has been removed from the database by a DELETE query. Therefore, it is a good practice to use a separate `EntityManager` for DELETE queries.

As with any operation that modifies the database, DELETE queries can only be executed within an active transaction and the changes are visible to other users (which use other `EntityManager` instances) only after `commit`.

Delete All Queries

The simplest form of a DELETE query removes all the instances of a specified entity class (including instances of subclasses) from the database.

For example, the following three equivalent queries delete all the `Country` instances:

```
DELETE FROM Country       // no variable
DELETE FROM Country c     // an optional variable
DELETE FROM Country AS c  // AS + an optional variable
```

ObjectDB supports using the `java.lang.Object` class in queries (as an extension to JPA), so the following query can be used to delete all the objects in the database:

```
DELETE FROM Object
```

DELETE queries are executed using the `executeUpdate` method:

```
  int deletedCount = em.createQuery("DELETE FROM Country").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.

On success - the `executeUpdate` method returns the number of objects that have been deleted by the query.

Selective Deletion

The structure of DELETE queries is very simple relative to the structure of SELECT queries. DELETE queries cannot include multiple variables and JOIN, and cannot include the GROUP BY, HAVING and ORDER BY clauses.

A WHERE clause, which is essential for removing selected entity objects, is supported.

For example, the following query deletes the countries with population size that is smaller than a specified limit:

```
DELETE FROM Country c WHERE c.population < :p
```

The query can be executed as follows:

```
Query query = em.createQuery(
    "DELETE FROM Country c WHERE c.population < :p");
int deletedCount = query.setParameter(p, 100000).executeUpdate();
```

4.2.7 UPDATE Queries in JPA/JPQL

Existing entity objects can be updated, as explained in chapter 2, by:

- Retrieving the entity objects into an `EntityManager`.
- Updating the relevant entity object fields within an active transaction.
- Applying changes to the database by calling the `commit` method.

JPQL UPDATE queries provide an alternative way for updating entity objects. Unlike SELECT queries, which are used to retrieve data from the database, UPDATE queries do not retrieve data from the database, but when executed, update the content of specified entity objects in the database.

Updating entity objects in the database using an UPDATE query may be slightly more efficient than retrieving entity objects and then updating them, but it should be used cautiously because bypassing the `EntityManager` may break its synchronization with the database. For example, the `EntityManager` may not be aware that a cached entity object in its persistence context has been modified by an UPDATE query. Therefore, it is a good practice to use a separate `EntityManager` for UPDATE queries.

As with any operation that modifies the database, UPDATE queries can only be executed within an active transaction and the changes are visible to other users (which use other `EntityManager` instances) only

after `commit`.

Update All Queries

The simpler form of UPDATE queries acts on all the instances of a specified entity class in the database (including instances of subclasses).

For example, the following three equivalent queries increase the population size of all the countries by 10%:

```
UPDATE Country SET population = population * 11 / 10
UPDATE Country c SET c.population = c.population * 11 / 10
UPDATE Country AS c SET c.population = c.population * 11 / 10
```

The UPDATE clause defines exactly one range variable (with or without an explicit variable name) for iteration. Multiple variables and JOIN are not supported. The SET clause defines one or more field update expressions (using the range variable name - if defined).

Multiple field update expressions, separated by commas, are also allowed. For example:

```
UPDATE Country SET population = 0, area = 0
```

UPDATE queries are executed using the `executeUpdate` method:

```
Query query = em.createQuery(
    "UPDATE Country SET population = 0, area = 0");
int updateCount = em.executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.

On success - the `executeUpdate` method returns the number of objects that have been modified by the query.

Selective Update

UPDATE queries cannot include the GROUP BY, HAVING and ORDER BY clauses, but the WHERE clause, which is essential for updating selected entity objects, is supported.

For example, the following query updates the population size of countries whose population size that is smaller than a specified limit:

```
UPDATE Country
SET population = population * 11 / 10
WHERE c.population < :p
```

The query can be executed as follows:

```
Query query = em.createQuery(
    "UPDATE Country SET population = population * 11 / 10 " +
    "WHERE c.population < :p");
int updateCount = query.setParameter(p, 100000).executeUpdate();
```

4.3 JPA Query Expressions (JPQL / Criteria)

Query expressions are the foundations on which JPQL and criteria queries are built.

Every query consists of clauses - SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY, and each clause consists of JPQL / Criteria query expressions.

Atomic Expressions

The atomic query expressions are:

- JPQL / Criteria Variables
- JPQL / Criteria Parameters
- JPQL / Criteria Literals

Every query expression consists of at least one atomic component. More complex query expressions are built by combining atomic expressions with operators and functions.

Operators and Functions

JPQL / Criteria queries support the following operators (in order of decreasing precedence):

- Navigation operator (.)
- Arithmetic operators:
  * (multiplication), / (division), + (addition) and - (subtraction).
- Comparison operators (including String and Collection operators):
  =, <>, <, <=,>, >=, IS [NOT] NULL, [NOT] BETWEEN,

[NOT] LIKE, [NOT] IN, IS [NOT] EMPTY, [NOT] MEMBER [OF]

- Logical operators: AND, OR, NOT.

In addition, JPA queries support predefined functions, which are also described in this section.

Organization of this Section

This section contains the following pages:

- Literals in JPQL and Criteria Queries
- Paths and Types in JPQL and Criteria API
- Numbers in JPQL and Criteria Queries
- Strings in JPQL and Criteria Queries
- Date and Time in JPQL and Criteria Queries
- Collections in JPQL and Criteria Queries
- Comparison in JPQL and Criteria API
- Logical Operators in JPQL and Criteria API

Detailed explanations on how to build criteria query expressions are provided as follows:

- Literals and Dates (`literal`, `nullLiteral`, `currentDate`, ...).
- Paths, navigation and types (`get`, `type`).
- Arithmetic expressions (`sum`, `diff`, `prod`, `quot`, `mod`, `abs`, `neg`, `sqrt`).
- String expressions (`like`, `length`, `locate`, `lower`, `upper`, `concat`, `substring, ...`).
- Data expressions (`currentDate`, `currentTime`, `currentTimestamp`).
- Collection expressions (`isEmpty`, `isNotEmpty`, `isMember`, `isNotMember`, `size`).
- Comparison expressions (`equal`, `notEqual`, `gt`, `ge`, `lt`, `le`, `between`, `isNull`, ...)
- Logical expressions (`and`, `or`, `not`, `isTrue`).

4.3.1 Literals in JPQL and Criteria Queries

Literals in JPQL, as in Java, represent constant values. JPQL supports various types of literals including `NULL`, boolean literals (`TRUE` and `FALSE`), numeric literals (e.g. `100`), string literals (e.g. `'abc'`), enum literals (e.g. `mypackage.MyEnum.MY_VALUE`) and entity type literals (e.g. `Country`).

JPQL literals should be used sparingly as queries that use parameters instead of literals are more generic and efficient because they can be compiled once and then run many times with different parameter values. Literals should only be embedded in JPQL queries when a single constant value is always used and never

replaced.

The NULL literal

> The NULL literal represents a null value, similarly to null in Java and SQL. Since JPQL is case insensitive, NULL, null and Null are equivalent. Notice that comparison with NULL in JPQL follows the SQL rules for NULL comparison rather than the Java rules, as explained in the Comparison Operators page.

Boolean Literals

Similarly to Java and SQL, JPQL supports two boolean literals - TRUE and FALSE. Since JPQL is case insensitive, TRUE is equivalent to true and True, and FALSE is equivalent to false and False.

Numeric Literals

JPQL supports the Java syntax as well as the SQL syntax for numeric literals. Numeric suffixes (e.g. 1.5F) are also supported. Following are examples of valid numeric literals in JPQL:

- int: 2010, -127, 0, 07777
- long: 2010L, -127L, 0L, 07777L
- float: 3.14F, 0f, 1e2f, -2.f, 5.04e+17f
- double: 3.14, 0d, 1e2D, -2., 5.04e+17

ObjectDB also supports hexadecimal numeric literals (e.g. 0xFF, 0xFFL) and octal numeric literals (e.g. 077, 077L), a feature that is not currently supported by all JPA implementations.

String Literals

JPQL follows the syntax of SQL for string literals in which strings are always enclosed in single quotes (e.g. 'Adam', '') and a single quote character in a string is represented by two single quotes (e.g. 'Adam''s').

ObjectDB also supports the syntax of Java and JDO for string literals in which strings are enclosed with double quotes (e.g. "Adam", "") and Java escape characters can be used (e.g. "Adam\'s", "abcd\n1234") but this is not supported by all the JPA implementations.

Unlike most other JPQL components, String literals (which represent data) are case sensitive, so 'abc' and 'ABC' are not equivalent.

Date and Time Literals

JPQL follows the syntax of SQL and JDBC for date literals:

- Date - {d 'yyyy-mm-dd'} - for example: {d '2010-12-31'}
- Time - {t 'hh:mm:ss'} - for example: {t '23:59:59'}
- Timestamp - {ts 'yyyy-mm-dd hh:mm:ss'} - for example: {ts '2011-01-03 13:59:59'}

Enum Literals

JPA 2 adds support for enum literals. Enum literals in JPQL queries use the ordinary Java syntax for enum values, but the fully qualified name of the enum type should always be specified.

For example, assuming we have the following enum definition:

```
package example.ui;

enum Color { RED, GREEN, BLUE }
```

Then `example.ui.Color.RED` is a valid literal in JPQL, but `Color.RED` is not.

Entity Type Literals

Entity type literals represent entity types in JPQL, similar to the way that `java.lang.Class` instances in Java represent Java types. Entity type literals have been added in JPA 2 to enable selective retrieval by type.

In JPQL an entity type literal is written simply as the name of the entity class (e.g. Country).
That is equivalent to `Country.class` in Java code. Notice that the name of the entity class is not enclosed in quotes (because type literals are not string literals).

By default, the name of an entity class is its unqualified name (i.e. excluding package name) but it can be modified by specifying another name explicitly in the @Entity's name annotation element.

Criteria Query Literals

The `CriteriaBuilder` interface provides two factory methods for building literal expressions.

Ordinary Literals

The main method, `literal`, takes a Java object and returns a literal expression. For example:

```
// Boolean literals:
Expression<Boolean> t = cb.literal(true);
Expression<Boolean> f = cb.literal(Boolean.FALSE);

// Numeric literals:
Expression<Integer> i1 = cb.literal(1);
Expression<Integer> i2 = cb.literal(Integer.ValueOf(2));
Expression<Double> d = cb.literal(3.4);

// String literals:
Expression<String> empty = cb.literal("");
Expression<String> jpa = cb.literal("JPA");

// Date and Time literals:
Expression<java.sql.Date> today = cb.literal(new java.sql.Date());
Expression<java.sql.Time> time = cb.literal(new java.sql.Time());
Expression<java.sql.Timestamp> now = cb.literal(new java.sql.Timestamp());

// Enum literal:
Expression<Color> red = cb.literal(Color.RED);

// Entity Type literal:
Expression<Class> type = cb.literal(MyEntity.class);
```

Null Literals

`Null` literal expressions can be built by the ordinary `literal` method:

```
Expression n = cb.literal(null);
```

or by a special `CriteriaBuilder`'s method, `nullLiteral`, that returns a typed expression:

```
Expression<String> strNull = cb.nullLiteral(String.class);
Expression<Integer> intNull = cb.nullLiteral(Integer.class);
```

4.3.2 Paths and Types in JPQL and Criteria API

Instances of user defined persistable classes (entity classes, mapped super classes and embeddable classes)

are represented in JPQL by the following types of expressions:

- Variables - FROM identification variables and SELECT result variables.
- Parameters - when instances of these classes are assigned as arguments.
- Path expressions that navigate from one object to another.

Instances of user defined persistable classes can participate in direct comparison using the = and <> operators. But more often they are used in JPQL path expressions that navigate to values of simple types (number, boolean, string, date).

Simple type values are more useful in queries. They have special operators and functions (e.g. for strings and for numbers), they can be compared by all six comparison operators, and they can be used in ordering.

Navigation through Path Expressions

A path expression always starts with an instance of a user defined class (represented by a variable, parameter or prefix path expression) and uses the dot (`.`) operator to navigate through persistent fields to other objects and values.

For example - `c.capital`, where c represents a Country entity object uses the `capital` persistent field in the `Country` class to navigate to the associated `Capital` entity object.

Path expression whose type is a persistable user class can be extended further by reusing the dot (.) operator. For example, `c.capital.name` is a nested path expression that continues from the Capital entity object to its name field. A path expression can be extended further only if its type is also a user defined persistable class. The dot (.) operator cannot be applied to collections, maps and values of simple types (number, boolean, string, date).

For a path expression to be valid the user defined persistable class must contain a persistent field (or property) with a matching name. The path expression, however, is valid even if the persistent field is declared as private (which is usually the case).

Navigation through a NULL value

The following query retrieves country names with their capital city names:

```
SELECT c.name, c.capital.name FROM Country c
```

The c identification variables is used for iteration over all the Country objects in the database.

For a country with no capital city, such as Nauru, `c.capital` is evaluated to NULL and c.capital.name is an attempt to navigate from a NULL value. In Java, a `NullPointerException` is thrown on any attempt to

access a field or a method via a null reference. In JPQL, the current FROM variable (or FROM tuple when there are multiple variables) is simply skipped. It might be easier to understand exactly how this works by considering the equivalent JOIN query.

Entity Type Expressions

The TYPE operator (which is new in JPA 2) returns the type of a specified argument, similarly to `java.lang.Object`'s `getClass` method in Java.

The following query returns the number of all the entity objects in the database, excluding Country entity objects:

```
SELECT COUNT(e) FROM Object e WHERE TYPE(e) <> Country
```

Binding an identification variable (e) to the Object class is an extension of ObjectDB that can be used to iterate over all the entity objects in the database. The `Country` literal represents the Country entity class. The TYPE operator returns the actual type of the iterated e. Only objects whose type is not Country are passed to the SELECT. The SELECT clause counts all these objects (this is an aggregate query with no GROUP BY - all the objects are considered as one group, and COUNT calculates its size).

Criteria Query Paths and Types

Paths and navigations are represented in the JPA Criteria API by the `Path` interface and by its subinterfaces (`From`, `Root`, `Join` and `Join`'s descendants).

Path Expressions

The `Root` and `Join` interfaces (which are subinterfaces of `From`) represent FROM variables. FROM variable expressions are considered as basic paths and also serve as starting point for building more complex paths through navigation.

Giving a `Path` instance, a child `Path` expression (which represents navigation from the parent path through a persistent field or property), can be constructed by the `get` method:

```
// FROM Variable Paths:
Root<Country> country = query.from(Country.class);
Join<Country, Country> neighborCountry = employee.join("neighbors");

// Navigation Paths:
```

```
Path<String> countryName = country.get("name");
Path<City> capital = country.get("capital");
Path<String> captialName = capital.get("name");
```

The path expressions in the above code can be divided into two main groups:

- FROM variable expressions, represented by subinterfaces of `From` (`Root`, `Join`) -
  The creation of a FROM expression automatically modifies the query by adding a variable to the FROM clause (representing iteration during query execution). The constructed variable expression can also be used explicitly in other query clauses.
- Navigation expressions, represented by the `Path` interface -
  The creation of a navigation path expression doesn't affect the built query directly. The constructed expression must be integrated into query clauses explicitly to have effect.

Type Expressions

Entity type expressions can be constructed by the `Path`'s `type` method. For example, the following criteria expression checks if the type of a specified entity `e` is not `Country`.

```
Predicate p = cb.notEqual(e.type(), cb.literal(Country.class));
```

In the above example, the comparison is between the type of the `e` object (which may represent any path including a root or a join) and the entity type `Country` (a criteria literal).

4.3.3 Numbers in JPQL and Criteria Queries

Numeric values may appear in JPQL queries in many forms:

- as numeric literals - e.g. `123`, `-12.5`.
- as parameters - when numeric values are assigned as arguments.
- as path expressions - in navigation to persistent numeric fields.
- as aggregate expressions - e.g. COUNT.
- as collection functions - when the return value is numeric, e.g. INDEX, SIZE.
- as string functions - when the return value is numeric, e.g. LOCATE, LENGTH.
- as composite arithmetic expressions that use operators and functions to combine simple numeric values into a more complex expression.

Arithmetic Operators

The following arithmetic operators are supported by JPA:

- 2 unary operators: + (plus) and - (minus).
- 4 binary operators: + (addition), - (subtraction), * (multiplication) and / (division).

ObjectDB also supports the % (modulo) and the ~ (bitwise complement) operators that are supported in Java and JDO. JPA follows Java numeric promotion principles. For example, the resulting type of a binary arithmetic operation on an `int` value and a `double` value is `double`.

The ABS Function

The ABS function removes the minus sign from a specified argument and returns the absolute value, which is always a positive number or zero.

For example:

- `ABS(-5)` is evaluated to `5`
- `ABS(10.7)` is evaluated to `10.7`

The ABS function takes as an argument a numeric value of any type and returns a value of the same type.

The MOD Function

The MOD function calculates the remainder of the division of one number by another, similar to the modulo operator (%) in Java (which is also supported by ObjectDB as an extension).

For example:

- `MOD(11, 3)` is evaluated to `2 (3 goes into 11 three times with a remainder of 2)`
- `MOD(8, 4)` is evaluated to `0 (4 goes into 8 twice with a remainder of 0)`

The MOD function takes two integer values of any type and returns an integer value. If the two operands share exactly the same type the result type is the same. If the two operands have different types, numeric promotion is used as with binary arithmetic operations in Java (e.g. for `int` and `long` operands the MOD function returns a `long` value).

The SQRT Function

The SQRT function returns the square root of a specified argument.

For example:

- `SQRT(9)` is evaluated to 3
- `SQRT(2)` is evaluated to `1.414213562373095`

The SQRT function takes as an argument a numeric value of any type and always returns a `double` value.

Criteria Query Arithmetic Expressions

JPQL arithmetic operators and functions (which are described above) are available also as JPA criteria query expressions. The `CriteriaBuilder` interface provides factory methods for building these expressions, as shown in the following examples.

Binary Operators

The creation of a binary arithmetic operator requires two operands. At least one operand must be a criteria numeric expression. The other operand may be either another numeric expression or a simple Java numeric object:

```
// Create path and parameter expressions:
Expression<Integer> path = country.get("population");
Expression<Integer> param = cb.parameter(Integer.class);

// Addition (+)
Expression<Integer> sum1 = cb.sum(path, param); // expression + expression
Expression<Integer> sum2 = cb.sum(path, 1000); // expression + number
Expression<Integer> sum3 = cb.sum(1000, path); // number + expression

// Subtraction (-)
Expression<Integer> diff1 = cb.diff(path, param); // expression - expression
Expression<Integer> diff2 = cb.diff(path, 1000); // expression - number
Expression<Integer> diff3 = cb.diff(1000, path); // number - expression

// Multiplication (*)
Expression<Integer> prod1 = cb.prod(path, param); // expression * expression
Expression<Integer> prod2 = cb.prod(path, 1000); // expression * number
Expression<Integer> prod3 = cb.prod(1000, path); // number * expression

// Division (/)
Expression<Integer> quot1 = cb.quot(path, param); // expression / expression
Expression<Integer> quot2 = cb.quot(path, 1000); // expression / number
Expression<Integer> quot3 = cb.quot(1000, path); // number / expression

// Modulo (%)
Expression<Integer> mod1 = cb.mod(path, param); // expression % expression
```

```
Expression<Integer> mod2 = cb.mod(path, 1000); // expression % number
Expression<Integer> mod3 = cb.mod(1000, path); // number % expression
```

The above examples demonstrate only integer expressions, but all the numeric types (`byte`, `short`, `int`, `long`, `float`, `double`, `BigInteger`, `BigDecimal`) are supported.

Unary Operators

Creation of the unary minus (`-`) operator and the ABS and SQRT functions requires one operand, which must be a numeric expression:

```
Expression<Integer> abs = cb.abs(param); // ABS(expression)
Expression<Integer> neg = cb.neg(path); // -expression
Expression<Integer> sqrt = cb.sqrt(cb.literal(100)); // SQRT(expression)
```

As shown above, a number can always be converted to a numeric expression by using the `literal` method.

4.3.4 Strings in JPQL and Criteria Queries

String values may appear in JPQL queries in various forms:

- as string literals - e.g. `'abc'`, ''.
- as parameters - when string values are assigned as arguments.
- as path expressions - in navigation to persistent string fields.
- as results of predefined JPQL string manipulation functions.

LIKE - String Pattern Matching with Wildcards

The [NOT] LIKE operator checks if a specified string matches a specified pattern. The pattern may include ordinary characters as well as the following wildcard characters:

- The percent character (%) - which matches **zero or more** of any character.
- The underscore character (_) - which matches any **single** character.

The left operand is always the string to check for a match (usually a path expression) and the right operand is always the pattern (usually a parameter or literal). For example:

- `c.name LIKE '_r%'` is TRUE for `'Brazil'` and FALSE for `'Denmark'`
- `c.name LIKE '%'` is always TRUE (for any `c.name` value).

- `c.name NOT LIKE '%'` is always `FALSE` (for any `c.name` value).

To match an actual underscore or percent character it has to be preceded by an escape character, which is also specified. For example:

- `'100%' LIKE '%\%' ESCAPE '\'` is evaluated to `TRUE`.
- `'100' LIKE '%\%' ESCAPE '\'` is evaluated to `FALSE`.

In the expressions above only the first percent character (`%`) is a wildcard. The second (which appears after the escape character) represents a real `%` character.

LENGTH - Counting Characters in a String

The `LENGTH(str)` function returns the number of characters in the argument string as an `int`.

For example:

- `LENGTH('United States')` is evaluated to `13`.
- `LENGTH('China')` is evaluated to `5`.

LOCATE - Locating Substrings

The `LOCATE(str, substr [, start])` function searches a substring and returns its position.

For example:

- `LOCATE('India', 'a')` is evaluated to `5`.
- `LOCATE('Japan', 'a', 3)` is evaluated to `4`.
- `LOCATE('Mexico', 'a')` is evaluated to `0`.

Notice that positions are one-based (as in SQL) rather then zero-based (as in Java). Therefore, the position of the first character is `1`. Zero (`0`) is returned if the substring is not found.

The third argument (when present) specifies from which position to start the search.

LOWER and UPPER - Changing String Case

The `LOWER(str)` and `UPPER(str)` functions return a string after conversion to lowercase or uppercase (respectively).

For example:

- `UPPER('Germany')` is evaluated to `'GERMANY'`.

- `LOWER('Germany')` is evaluated to `'germany'`.

TRIM - Stripping Leading and Trailing Characters

The `TRIM([[LEADING|TRAILING|BOTH] [char] FROM] str)` function returns a string after removing leading and/or trailing characters (usually space characters).

For example:

- `TRIM(' UK ')` is evaluated to `'UK'`.
- `TRIM(LEADING FROM ' UK ')` is evaluated to `'UK '`.
- `TRIM(TRAILING FROM ' UK ')` is evaluated to `' UK'`.
- `TRIM(BOTH FROM ' UK ')` is evaluated to `'UK'`.

By default, space characters are removed, but any other character can also be specified:

- `TRIM('A' FROM 'ARGENTINA')` is evaluated to `'RGENTIN`.
- `TRIM(LEADING 'A' FROM 'ARGENTINA')` is evaluated to `'RGENTINA'`.
- `TRIM(TRAILING 'A' FROM 'ARGENTINA')` is evaluated to `'ARGENTIN'`.

CONCAT - String Concatenation

The `CONCAT(str1, str2, ...)` function returns the concatenation of the specified strings.

For example:

- `CONCAT('Serbia', ' and ', 'Montenegro')` is evaluated to `'Serbia and Montenegro'`.

SUBSTRING - Getting a Portion of a String

The `SUBSTRING(str, pos [, length])` function returns a substring of a specified string.

For example:

- `SUBSTRING('Italy', 3)` is evaluated to `'aly'`.
- `SUBSTRING('Italy', 3, 2)` is evaluated to `'al'`.

Notice that positions are one-based (as in SQL) rather then zero based (as in Java). If length is not specified (the third optional argument), the entire string suffix, starting at the specified position, is returned.

Java String Methods (ObjectDB Extension)

ObjectDB also supports ordinary Java String methods.

For example:

- `'Canada'.length()` is evaluated to `6`.
- `'Poland'.toLowerCase()` is evaluated to `'poland'`.

The `matches` method of the `String` class can be useful when there is a need for pattern matching using regular expressions (which are more powerful than the LIKE operator).

Criteria Query String Expressions

JPQL string operators and functions (which are described above) are available also as JPA criteria query expressions. The `CriteriaBuilder` interface provides factory methods for building these expressions, as shown in the following examples:

```
// Create path and parameter expressions:
Expression<String> path = country.get("name");
Expression<String> param = cb.parameter(String.class);

// str [NOT] LIKE pattern
Predicate l1 = cb.like(path, param);
Predicate l2 = cb.like(path, "a%");
Predicate l3 = cb.notLike(path, param);
Predicate l4 = cb.notLike(path, "a%");
  // additional methods take also an escape character

// LENGTH(str)
Expression<Integer> length = cb.length(path);

// LOCATE(str, substr [, start])
Expression<Integer> l1 = cb.locate(path, param);
Expression<Integer> l2 = cb.locate(path, "x");
Expression<Integer> l3 = cb.locate(path, param, cb.literal(2));
Expression<Integer> l4 = cb.locate(path, "x", 2);

// LOWER(str) and UPPER(str)
Expression<String> lower = cb.lower(path);
Expression<String> upper = cb.upper(param);

// TRIM([[LEADING|TRAILING|BOTH] [char] FROM] str)
Expression<String> t1 = cb.trim(path);
```

```
Expression<String> t2 = cb.trim(literal(&#39; &#39;), path);
Expression<String> t3 = cb.trim(&#39; &#39;, path);
Expression<String> t4 = cb.trim(Trimspec.BOTH, path);
Expression<String> t5 = cb.trim(Trimspec.LEADING, literal(&#39; &#39;), path);
Expression<String> t6 = cb.trim(Trimspec.TRAILING, &#39; &#39;, path);

// CONCAT(str1, str2)
Expression<String> c1 = cb.concat(path, param);
Expression<String> c2 = cb.concat(path, ".");
Expression<String> c3 = cb.concat("the", path);

// SUBSTRING(str, pos [, length])
Expression<String> s1 = cb.substring(path, cb.literal(2));
Expression<String> s2 = cb.substring(path, 2);
Expression<String> s3 = cb.substring(path, cb.literal(2), cb.literal(3));
Expression<String> s4 = cb.substring(path, 2, 3);
```

As demonstrated above, most methods are overloaded in order to support optional arguments and when applicable simple Java objects as well as criteria expressions.

4.3.5 Date and Time in JPQL and Criteria Queries

Date and time expressions may appear in JPQL queries:

- as date and time literals - e.g. `{d '2011-12-31'}`, `{t '23:59:59'}`.
- as parameters - when date and time values are assigned as arguments.
- as path expressions - in navigation to persistent date and time fields.
- as results of predefined JPQL current date and time functions.

Current Date and Time

JPA defines special JPQL expressions that are evaluated to the date and time on the database server when the query is executed:

- `CURRENT_DATE` - is evaluated to the current date (a `java.sql.Date` instance).
- `CURRENT_TIME` - is evaluated to the current time (a `java.sql.Time` instance).
- `CURRENT_TIMESTAMP` - is evaluated to the current timestamp, i.e. date and time (a `java.sql.Timestamp` instance).

Extracting Date Parts

JPA doesn't define standard methods for extracting date and time parts but some JPA implementations, as well as ObjectDB, support such functions as an extension. ObjectDB supports 6 functions for extracting the `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` and `SECOND`.

For example:

- `YEAR({d '2011-12-31'})` is evaluated to `2011`.
- `MONTH({d '2011-12-31'})` is evaluated to `12`.
- `DAY({d '2011-12-31'})` is evaluated to `31`.
- `HOUR({t '23:59:00'})` is evaluated to `23`.
- `MINUTE({t '23:59:00'})` is evaluated to `59`.
- `SECOND({t '23:59:00'})` is evaluated to `0`.

Date and Time in Criteria Queries

The `CriteriaBuilder` interface provides three factory methods for building date and time expressions that represent the current date and/or time:

```
// Create current date expression:
Expression<javax.sql.Date> date = cb.currentDate(); // date only

// Create current time expression:
Expression<javax.sql.Time> time = cb.currentTime(); // time only

// Create current date &amp;amp; time expression:
Expression<javax.sql.Timestamp> ts = cb.currentTimestamp(); // both
```

Unlike constant date literals which are built once on the client side, the current date and time expressions are reevaluated on the server on every query execution to reflect the date and time when the query is run.

Functions for extracting date and time parts are also available in criteria queries by using the generic `CriteriaBuilder`'s `function` method, as follow:

```
// Create expressions that extract date parts:
Expression<Integer> year = cb.function("year", Integer.class, date);
Expression<Integer> month = cb.function("month", Integer.class, date);
Expression<Integer> day = cb.function("day", Integer.class, ts);

// Create expressions that extract time parts:
```

```
Expression<Integer> hour = cb.function("hour", Integer.class, time);
Expression<Integer> minute = cb.function("minute", Integer.class, time);
Expression<Integer> second = cb.function("second", Integer.class, ts);
```

4.3.6 Collections in JPQL and Criteria Queries

Collections may appear in JPQL queries:

- as parameters - when collections are assigned as arguments.
- as path expressions - in navigation to persistent collection fields.

IS [NOT] EMPTY

The IS [NOT] EMPTY operator checks whether a specified collection is empty or not.

For example:

- `c.languages IS EMPTY` is TRUE if the collection is empty and FALSE otherwise.
- `c.languages IS NOT EMPTY` is FALSE if the collection is empty and TRUE otherwise.

SIZE

The SIZE(collection) function returns the number of elements in a specified collection.

For example:

- `SIZE(c.languages)` is evaluated to the number of languages in that collection.

[NOT] MEMBER [OF]

The `[NOT] MEMBER OF` operator checks if a specified element is contained in a specified persistent collection field.

For example:

- `'English' MEMBER OF c.languages` is TRUE if `languages` contains `'English'` and FALSE if not.
- `'English' NOT MEMBER OF c.languages` is TRUE if `languages` does not contain `'English'`.

[NOT] IN

The `[NOT] IN` operator provides an additional method for checking if a specified element is contained in a

collection.

JPA distinguishes between the `MEMBER OF` operator, which should be used for checking a collection field, and the `IN` operator, which should be used for checking other collections, such as a collection that is passed to the query as a parameter.

For example:

- `'English' IN :languages` is TRUE if the argument for the `languages` parameter is a collection that contains `'English'` and FALSE if not.
- `'English' NOT IN :languages` is TRUE if the argument for the `languages` parameter is a collection that doesn't contain `'English'`.

ObjectDB enables as an extension to standard JPQL to use both operators (`IN` and `MEMBER OF`) with any type of collection, so in ObjectDB these operators are treated as synonyms.

Criteria Query Collection Expressions

JPQL collection operators and functions (which are described above) are available also as JPA criteria query expressions. The `CriteriaBuilder` interface provides factory methods for building these expressions, as shown in the following examples:

```
// Create path and parameter expressions:
Expression<Collection<String>> languages = country.get("languages");
Expression<String> param = cb.parameter(String.class);

// collection IS [NOT] EMPTY
Predicate e1 = cb.isEmpty(languages);
Predicate e2 = cb.isNotEmpty(languages);

// element [NOT] MEMBER OF collection
Predicate m1 = cb.isMember(param, languages);
Predicate m2 = cb.isMember("English", languages);
Predicate m3 = cb.isNotMember(param, languages);
Predicate m4 = cb.isNotMember("French", languages);

// element [NOT] IN collection:
Predicate i1 = param.in(languages);
Predicate i2 = param.in("English", "French");

// SIZE(collection)
```

```
Expression<Integer> size = cb.size(languages);
```

4.3.7 Comparison in JPQL and Criteria API

Most JPQL queries use at least one comparison operator in their WHERE clause.

Comparison Operators

ObjectDB supports two sets of comparison operators, as shown in the following table:

	Set 1 - JPQL / SQL	Set 2 - Java / JDO
Less Than	<	<
Greater Than	>	>
Less Than or Equal To	<=	<=
Greater Than or Equal To	>=	>=
Equal	=	==
Not Equal	!=	<>

The two sets differ in the Equal and the Not Equal operators. JPQL follows the SQL notation, where Java uses its own notation (which is also in use by JDOQL, the JDO Query Language). ObjectDB supports both forms. Besides the different notation, there is also a difference in the way that NULL values are handled by these operators.

Comparing NULL values

The following table shows how NULL values are handled by each comparison operator. One column presents comparison of NULL value with a non NULL value. The other column presents comparison of two NULL values:

Operators	One NULL operand	Two NULL operands
<, <=, >, >=	NULL	NULL
=	NULL	NULL
<>	NULL	NULL
==	FALSE	TRUE
!=	TRUE	FALSE

Comparison operators are always evaluated to TRUE, FALSE or NULL.

When both operands are not NULL (not shown in the table) the operator is evaluated to either TRUE or FALSE, and in that case, == is equivalent to = and != is equivalent to <>.

When at least one of the two operands is NULL, == and != implement the ordinary Java logic, in which, for example, `null == null` is `true`. All the other operators implement the SQL logic in which NULL represents an unknown value and expressions that include an unknown value are evaluated as unknown, i.e. to NULL.

IS [NOT] NULL

To check for NULL using standard JPQL you can use the special `IS NULL` and `IS NOT NULL` operators which are provided by JPQL (and SQL):

```
c.president IS NULL
c.president IS NOT NULL
```

The expressions above are equivalent (respectively) to the following non standard JPQL (but standard Java and JDOQL) expressions:

```
c.president == null
c.president != null
```

Comparable Data Types

Comparison is supported for values of the following data types:

- Values of numeric types, including primitive types (`byte`, `short`, `char`, `int`, `long`, `float`, `double`), wrapper types (`Byte`, Short, Character, `Integer`, `Long`, `Float`, `Double`), `BigInteger` and `BigDecimal` can be compared by using any comparison operator.
- String values can be compared by using any comparison operator.
  Equality operators (=, <>, ==, !=) on strings in queries follow the logic of Java's `equals`, comparing the content rather than the identity.
- Date values can be compared by using any comparison operator.
  Equality operators (=, <>, ==, !=) on date values in queries follow the logic of `equals`, comparing the content rather than the identity.
- Values of the `boolean` and `Boolean` types can be compared by equality operators (=, <>, ==, !=) which follow the logic of Java's `equals` (for `Boolean` instances).
- Enum values can be compared by using the equality operators (=, <>, ==, !=).
- Instances of user defined classes (entity classes and embeddable classes) can be compared by using

the equality operators (=, <>, ==, !=). For entities, e1 = e2 if e1 and e2 have the same type and the same primary key value. For embeddable objects, e1 = e2 if e1 and e2 have exactly the same content.

ObjectDB supports comparison of any two values that belong to the same group as detailed above. Therefore, for example, a `double` value can be compared to a `BigInteger` instance but not to a `String` instance.

[NOT] BETWEEN

The BETWEEN operator is a convenient shortcut that can replace two simple comparisons.

The two following expressions are equivalent (`:min` and `:max` are query parameters):

```
x BETWEEN :min AND :max

x >= :min AND x <= :max
```

Similarly, NOT BETWEEN is also a shortcut and the following expressions are equivalent:

```
x NOT BETWEEN :min AND :max

x < :min OR x > :max
```

Criteria Query Comparison

JPQL comparison operators (which are described above) are available also as JPA criteria query expressions. The `CriteriaBuilder` interface provides factory methods for building these expressions, as shown in the following examples:

```
// Create String path and parameter expressions:
Expression<String> name = country.get("name");
Expression<String> nameParam = cb.parameter(String.class);

// Create Integer path and parameter expressions:
Expression<Integer> area = country.get("area");
Expression<Integer> areaParam = cb.parameter(Integer.class);

// Equal (=)
Predicate eq1 = cb.equal(name, nameParam);
Predicate eq2 = cb.equal(name, "India");
```

```
Predicate eq3 = cb.equal(area, areaParam);
Predicate eq4 = cb.equal(area, 1000000);

// Not Equal (<>)
Predicate ne1 = cb.notEqual(name, nameParam);
Predicate ne2 = cb.notEqual(name, "India");
Predicate ne3 = cb.notEqual(area, areaParam);
Predicate ne4 = cb.notEqual(area, 1000000);

// Greater Than (>)
Predicate gt1 = cb.greaterThan(name, nameParam);
Predicate gt2 = cb.greaterThan(name, "India");
Predicate gt3 = cb.gt(area, areaParam);
Predicate gt4 = cb.gt(area, 1000000);

// Greater Than or Equal (>=)
Predicate ge1 = cb.greaterThanOrEqualTo(name, nameParam);
Predicate ge2 = cb.greaterThanOrEqualTo(name, "India");
Predicate ge3 = cb.ge(area, areaParam);
Predicate ge4 = cb.ge(area, 1000000);

// Less Than (<)
Predicate lt1 = cb.lessThan(name, nameParam);
Predicate lt2 = cb.lessThan(name, "India");
Predicate lt3 = cb.lt(area, areaParam);
Predicate lt4 = cb.lt(area, 1000000);

// Less Than or Equal (<=)
Predicate le1 = cb.lessThanOrEqualTo(name, nameParam);
Predicate le2 = cb.lessThanOrEqualTo(name, "India");
Predicate le3 = cb.le(area, areaParam);
Predicate le4 = cb.le(area, 1000000);

// BETWEEN
Predicate b1 = cb.between(name, nameParam, cb.literal("Y"));
Predicate b2 = cb.between(name, "X", "Y");
Predicate b3 = cb.between(area, areaParam, cb.literal(2000000));
Predicate b4 = cb.between(area, 1000000, 2000000);

// IS [NOT] NULL
Predicate n1 = cb.isNull(name);
Predicate n2 = cb.isNotNull(name);
```

As demonstrated above, the first argument of every one of these methods is a criteria expression. The

second argument (and the third argument in `between`) can be either a criteria expression or a comparable Java object.

The 2 letter methods (`gt`, `ge`, `lt`, `le`) can only be used for numeric comparison. The other methods can be used with any comparable objects (and `isNull` and `isNotNull` also take a non comparable object as an argument). For comparison of numbers, `gt` and `greaterThan` are equivalent, but it is a good practice to use the short form (`gt`) when applicable to emphasis a numeric comparison.

4.3.8 Logical Operators in JPQL and Criteria API

Logical operators in JPQL and in JPA criteria queries enable composition of complex JPQL boolean expressions out of simple JPQL boolean expressions.

Logical Operators

ObjectDB supports 2 sets of logical operators, as shown in the following table:

Set 1 - JPQL / SQL	Set 2 - Java / JDO
AND	&&
OR	||
NOT	!

JPQL follows the SQL notation, while Java uses its own notation (which is also in use by JDOQL, the JDO Query Language). ObjectDB supports both forms.

Binary AND (&&) Operator

The following query retrieves countries whose population **and** area (both) exceed specified limits:

```
SELECT c FROM Country c
WHERE c.population > :population AND c.area > :area
```

A valid operand of an `AND` operator must be one of: `TRUE`, `FALSE`, and `NULL`.

The following table shows how the `AND` operator is evaluated based on its two operands:

	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE

NULL	NULL	FALSE	NULL

NULL represents unknown. Therefore, if one operand is NULL and the other operand is FALSE the result is FALSE, because one FALSE operand is sufficient for a FALSE result. If one operand is NULL and the other operand is either TRUE or NULL, the result is NULL (unknown).

ObjectDB supports the Java/JDO && operator as a synonym of AND as part of its JDO support.

Binary OR (||) Operator

The following query retrieves countries whose population **or** area exceeds a specified limit:

```
SELECT c FROM Country c
WHERE c.population > :population OR c.area > :area
```

A valid operand of an OR operator must be one of: TRUE, FALSE, and NULL.

The following table shows how the OR operator is evaluated based on its two operands:

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NULL represents unknown. Therefore, if one operand is NULL and the other operand is TRUE the result is TRUE, because one TRUE operand is sufficient for a TRUE result. If one operand is NULL and the other operand is either FALSE or NULL, the result is NULL (unknown).

ObjectDB supports the Java/JDO || operator as a synonym for OR as part of its JDO support.

Unary NOT (!) Operator

The following query retrieves all the countries whose population does not exceed a specified limit:

```
SELECT c FROM Country c
WHERE NOT (c.population > :population)
```

The operand of a NOT operator must be one of: TRUE, FALSE, or NULL.

The following table shows how the NOT operator is evaluated based on its operand:

TRUE	**FALSE**	**NULL**

FALSE	TRUE	NULL

If the operand is NULL, which represents unknown, the result is also NULL (unknown).

ObjectDB also supports the Java/JDO ! operator as a replacement for NOT as part of its JDO support.

Criteria Query Logical Operators

Boolean Expressions and Predicates

Boolean expressions are represented in criteria queries by Expression<Boolean> and descendant interfaces. For example, a boolean path (a field or a property) is represented by Path<Boolean>:

```
Path<Boolean> isInUN = country.get("isInUN");
Path<Boolean> isInEU = country.get("isInEU");
Path<Boolean> isInOECD = country.get("isInOECD");
```

Predicate is a special sub interface of Expression<Boolean> that represents many operator and function expressions whose type is boolean - such as comparison operators:

```
Predicate isLarge = cb.gt(country.get("area"), 1000000);
```

AND / OR Expressions

The CriteriaBuilder interface provides factory methods that take two Expression<Boolean> operands (including Predicate instances) and return a new Predicate instance:

```
Predicate p1 = cb.and(isInUN, isInEU);  // Member of both UN and EU
Predicate p2 = cb.or(isInOECD, isLarge); // Either OECD member or large
```

Additional factory methods are available for a variant number of predicates:

```
Predicate p3 = cb.and(p1, isLarge, cb.isTrue(isInOECD));
Predicate p4 = cb.or(p2, cb.isTrue(isInUN), cb.isTrue(isInEU));
```

In the above code non Predicate boolean expressions are converted to Predicate instances using the isTrue method. This is required because in the non binary version the factory methods accept only

Predicate instances as arguments.

NOT Expression

There are two ways to create a NOT operator:

```
Predicate p5 = cb.not(isInUN);
Predicate p6 = isLarge.not();
```

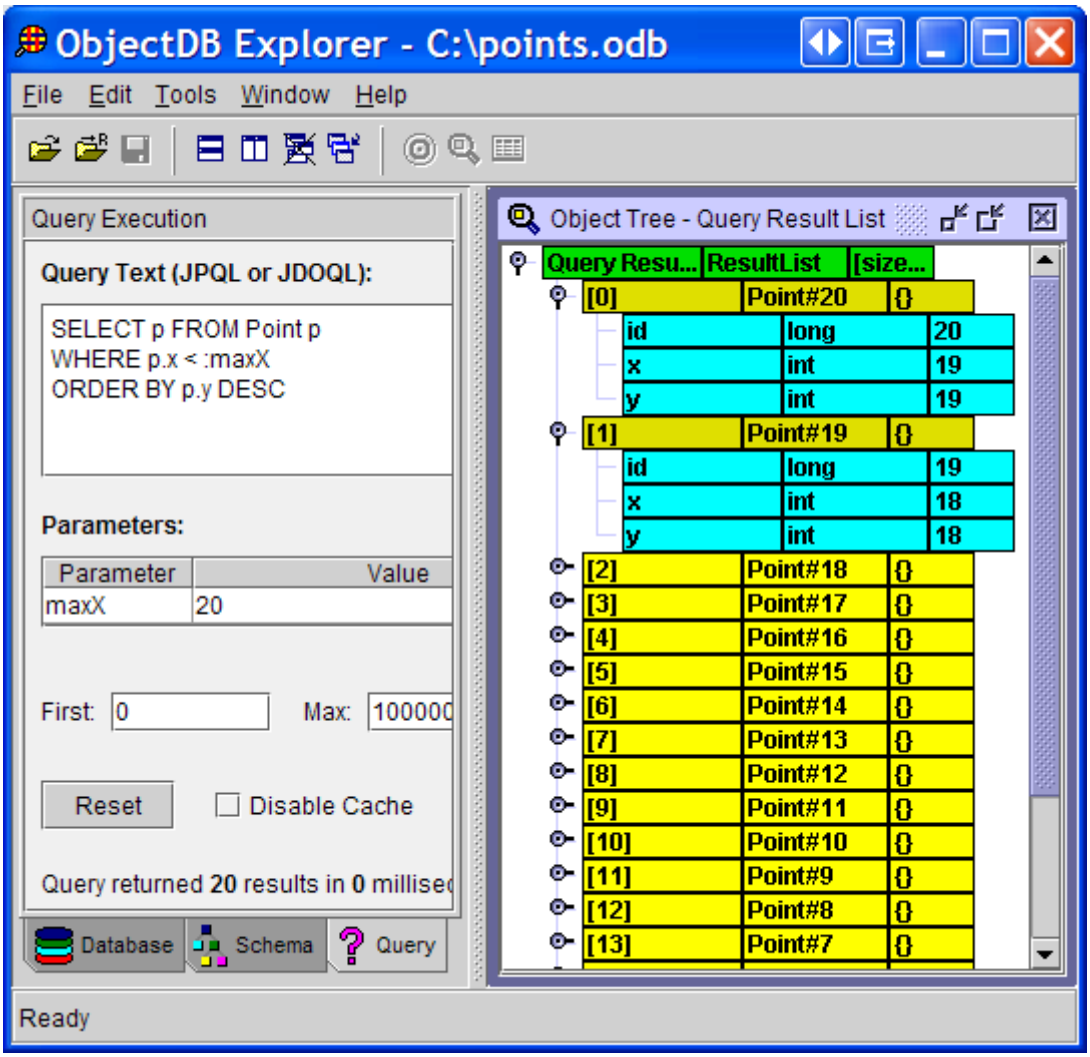
The CriteriaBuilder's not method creates a Predicate by negation of a specified boolean expression. Alternatively, to create a negation of a Predicate instance, the Predicate's not method can be invoked.

Chapter 5 - Database Tools and Utilities

This chapter explains how to use the following ObjectDB tools and utilities:

5.1 Database Explorer

ObjectDB Database Explorer is a visual GUI tool for managing ObjectDB databases. It can be used to view data in ObjectDB databases, execute JPQL and JDOQL queries and edit the content of databases.



Running the Explorer

The ObjectDB Explorer is contained in the `explorer.jar` executable jar file, which is located in the `bin` directory of ObjectDB. It depends on the `objectdb.jar` file.

You can run it from the command line as follows:

```
> java -jar explorer.jar
```

If `explorer.jar` is not in the current directory a path to it has to be specified.

Alternatively, you can run the Explorer by double clicking `explorer.jar` or by running `explorer.exe` (on Windows) or `explorer.sh` (on Unix/Linux, after editing the shell file, setting the paths to the `objectdb.jar` file and to the JVM).

Connecting to the Database

Opening a Database Connection

To open a local database file in embedded mode use the **File > Open Embedded...** command and select a local database file.

To open a database in client-server mode select **File > Open C/S Connection...** and provide host, port, username and password for a client-server connection. You also have to specify a database path on the server, possibly using the **Browse** button.

Recently used databases can also be opened using the **File > Recent Connections** command. By default, when the Explorer starts it tries to open a connection to the most recently used database automatically.

Tabbed Windows

Three tabbed windows are displayed on the left side of the Explorer when a database is open:

- The [Database] window is split into two sub windows. The top sub window displays general information about the database and the bottom sub window displays a list of bookmarked entity objects.
- The [Schema] window shows the user defined persistable types (entity and embeddable classes) in the database and their persistent fields and indexes.
- The [Query] window enables running JPQL and JDOQL queries, as discussed below.

Closing a Database Connection

Use the **File > Close** menu command to close a currently open database connection.

Viewing Database Content

Viewer Windows

The Explorer provides two types of viewer windows for viewing the database data.

The **Table** window follows the approach of traditional visual database tools. Every row in the table represents a single object, every column represents a persistent field, and the content of a cell is the value of a single field in a single database object. This type of viewer is useful for viewing data of a simple object model.

In most cases, the **Tree** window, which is designed to handle more complex object models, is preferred. A Tree window displays objects as a tree. Every database object is represented by a tree node and the values

of its persistent fields are represented as child nodes. Tree windows provide easier navigation. Because every reference between two database objects is represented by a parent-child relationship in the tree, you can navigate among objects by expanding nodes in the tree, similar to exploring objects in a visual debugger. Notice that the same database object may be accessed using different paths in the tree and therefore may be represented by more than one node. To help identify circles in the graph of objects, a special {R} sign (indicating recursive) is displayed for an object that is already shown in a higher level of the same tree path (i.e. the object is a descendant of itself in the tree).

To open a new viewer window either write a query in the [Query] tabbed window or select an element for viewing (an entity class in the [Schema] tabbed window, a bookmark in the [Database] tabbed window or an object in a currently open Table or Tree window). You can open a new viewer window using the **Window > Open Tree Window** and **Window > Open Table Window** commands. When the target element is an object in an open viewer window the **Window > Focus Selection** command switches the current viewer to focus on the selected object.

Executing Queries

The `[Query]` window on the left side enables execution of JPQL and JDOQL queries.

To execute a query:

- Enter a query string.
- In the [Parameters] table provide arguments for parameters (if any). An entity object can be specified by type and primary key separated by # (e.g. Point#1). A collection can be specified as a comma separated list of elements.
- Use the [First] and [Max] fields to set the result range.
- Check [Disable Cache] to bypass query program and result caches.
- Click the **Execute** button to run the query.

If the query is valid, a default viewer (a Tree window by default) is opened with the query results and the size of the result collection and the query execution time are displayed on the query window. If the query compilation fails, an error is displayed on the query window and no viewer window is opened. You can also execute a query using the **Window > Open Tree Window** or the **Window > Open Table Window** commands in order to specify a preferred viewer window type for the results.

On the bottom of the query window there is a log panel that displays the selected query execution plan. This is useful, for example, for checking which indexes are used.

The **Reset** button clears the content of all the fields.

Refreshing the Cache

When a database is open in the Explorer in client server mode, it can be accessed simultaneously by other applications. If the database is modified by another process the viewer windows in the Explorer might display cached content that does not reflect the up to date data in the database. In this case you can refresh the cache and the viewer windows using the **File > Refresh Data** menu command.

Using Bookmarks

To bookmark an entity object for easier future access - first select it in a viewer window and then select the **Tools > Bookmark...** command, specify a bookmark name and click **OK**. Bookmarked entity objects can be accessed from the [Database] tabbed window.

Editing Database Content

The Explorer is used mainly as a viewer of databases but it can also function as an editor.

New Entity Objects

To construct new entity objects and store them in the database, open the [Create New Entity Objects] dialog box using the **Edit > New Entity Objects...** command. In the dialog box select the entity class and specify the number of objects to construct. Click the **Create and Persist** button to construct and store the new entity objects.

Editing Values and References

You can edit simple field values inline in the Tree and Table viewer windows. Editing is started by double clicking the field, pressing F2, or even by simply typing the new value. Use the **Edit > Edit Multi Line String** command to edit a multi line string.

Reference fields can be edited by using the following commands:

- The **Edit > Set Reference > Set to Null** command is used to set a reference to `null`.
- The **Edit > Set Reference > Set to Existing Entity...** command is used to set a reference to an existing entity object, which has to be specified by type and primary key separated by # (e.g. `Point#1`).
- The **Edit > Set Reference > Set to New Object...** command is used to set a reference to a new object (not necessarily an entity object).

Field values can also be set using standard clipboard commands (**Cut**, **Copy** and **Paste**). For example, an entity object can be selected and copied as a reference into the clipboard (in any viewer window) and

then pasted on a reference that has to be set.

The functionality of the **Edit > Delete** command depends on the context. Deleting a reference field sets the value to null without deleting any referenced entity object. On the other hand, deleting an entity object that is represented by a child of an Extent node in a viewer window deletes the object itself from the database.

Editing Collections

Elements can be added to collections by using the following commands:

- The **Edit > Add to Collection > Add Null** command is used to add a `null` value.
- The **Edit > Add to Collection > Add Existing Entity...** command is used to add a reference to an existing entity object, which has to be specified by type and primary key separated by # (e.g. Point#1).
- The **Edit > Add to Collection > Add New Object...** command is used to add a reference to a new object (not necessarily an entity object).

Elements can be removed from the collection using the **Edit > Delete** command.

The order of elements in an ordered collection can be modified by using the **Edit > Move Element** commands.

Saving Changes

The Explorer manages an active transaction for every open database file. All the editing operations are associated with the active transaction. The **File > Save** menu command commits the transaction (applying all the changes to the database). The **File > Discard Changes** menu command rolls back the transaction (discarding all the changes). After **File > Save** and **File > Discard Changes**, the Explorer automatically begins a new transaction for the next editing session.

Options and Settings

The Explorer settings are organized in the [Options] dialog box, which can be opened using the **Tools > Options...** menu command.

The [Options] dialog box contains three tabbed pages: [General], [Fonts] and [Views].

The [General] Page

The [General] page contains the following settings:

- The [Encoding] combo box is useful when strings in the database are not encoded using Unicode. ObjectDB stores `String` instances in the database using the same encoding that they have in memory,

which is usually Unicode. If you store String instances in the database that have a different memory representation you have to set that encoding to manage these strings in the Explorer. This setting is relevant only in the Explorer. In your applications - retrieved String instances always have the same encoding as they had when they have been persisted.

- The [Automatically open recently used database at startup] check box specifies whether or not to open the last opened database when the Explorer starts.
- The [Table (instead of Tree) as a default viewer window] check box specifies if Table windows should used by default rather than Tree windows.
- The [Darker colors for open branches in Tree windows] check box specifies whether or not to use different colors for open and close nodes in Tree windows.
- The [Classpath for persistent classes and metadata] field specifies a path to locate persistent classes and XML metadata. Setting this field is not mandatory because you can browse and edit ObjectDB database files when class and metadata files are not available. Some features of the Explorer, however, do require setting the classpath. For instance, executing queries that include user defined methods can be supported by the Explorer only when the code of these methods is available using the specified classpath.

The [Fonts] Page

The [Fonts] page is used to set the appearance of different Explorer components. Select one or more elements on the left side and then use the combo boxes on the right side to choose font face, font size, font style, background color and foreground color. Click the **Reset** button to apply the default settings to the selected elements. To discard all changes and apply the default settings to all the elements, click the **Reset All** button.

The [Views] Page

In the [Views] page you can select the persistent fields that are shown in the viewer windows and their order of appearance. This may be useful when working with classes with a large number of fields where displaying all the fields is problematic (for instance, in a row in the table viewer).

There are three views. The [Table View] determines which fields in each persistent class are displayed as columns in Table viewer windows. The [Tree View] determines which fields are displayed as child nodes when browsing persistent objects using the Tree viewer windows. The [Summary] view is also used for the Tree viewer. It determines which fields are displayed as a summary of a persistent object in the node that represents the object itself. The [Table View] and the [Tree View] are initialized to contain all the persistent fields, but the [Summary] view is initialized as an empty view. Therefore, unless set, persistent objects are presented in the Tree viewer as an empty set of fields (using "{}").

To set a view for a class, first select one of the three supported views and then select the persistable class in the list of classes. You can change the visibility of persistent fields of the class (in the selected view) using the Left and Right arrow buttons to move fields between the [Shown Fields] and the [Hidden Fields] lists. You can change the order of the shown fields by using the Up and Down arrow buttons or the [Field Ordering] combo box. A view for a class can also be set by right clicking one of its instances in the a viewer window (Table or Tree) and using the [Set View] context menu command.

5.2 Database Server

ObjectDB Server is a tool that manages ObjectDB databases in a separate dedicated process, making these databases accessible to client applications in other processes including ones on other remote machines.

The main benefits in running an ObjectDB server and using the client-server mode are:

- The ability to access and use databases from different processes simultaneously.
- The ability to access and use databases on remote machines over the network.

Since client-server mode carries the overhead of TCP/IP communication between the client and the server it is usually slower than embedded mode. In embedded mode, ObjectDB is integrated as a library and runs within the application process, which is much more efficient. As a result, embedded mode should be preferred when possible. For example, if an ObjectDB database is accessed directly only by a web application, it should be embedded in that web application and run within the web server process.

Starting the ObjectDB Server

The ObjectDB Server tool is bundled in the `objectdb.jar` file.

You can run it from the command line as follows:

```
> java -cp objectdb.jar com.objectdb.Server
```

If `objectdb.jar` is not in the current directory a path to it has to be specified.

Running the server with no arguments displays the following usage message:

```
ObjectDB Server [version 2.0]
Copyright (c) 2010, ObjectDB Software, All rights reserved.

Usage: java com.objectdb.Server [options] start | stop | restart
```

```
options include:
 -conf <path>  :  specify a configuration file explicitly
 -port <port>  :  override configuration's server port
 -silent       :  avoid printing messages to the standard output
 -console      :  avoid showing the server tray icon
```

To start the server, use the **start** command line argument:

```
> java com.objectdb.Server start
```

The Server configuration is loaded automatically as explained in chapter 6. This can be overridden by specifying a configuration path explicitly on the command line:

```
> java com.objectdb.Server -conf my_objectdb.conf start
```

The TPC/IP port on which the server listens for new connections is also specified in the Server Configuration, but can be overridden by an explicit command line option:

```
> java com.objectdb.Server -port 8888 start
```

You can also use standard JVM arguments. For instance, you can increase the maximum JVM heap size and improve performance by using HotSpot JVM server mode:

```
> java -server -Xmx512m com.objectdb.Server start
```

Stopping and Restarting

To stop the server you can use the **stop** command:

```
> java com.objectdb.Server stop
```

To stop the server and then immediately start it again, use the **restart** command:

```
> java com.objectdb.Server restart
```

While an ObjectDB Server is running in the foreground the command window may be blocked. Therefore, you may need a new command window for the **stop** and **restart** commands.

The `-conf` and `-port` options can also be used with the **stop** and **restart** commands.

Running the Server on Unix

On Unix you can use a shell script to run and manage the server. A sample script, `server.sh`, is included in the `bin` directory. To use that script you have to edit the paths to the `objectdb.jar` file and to the JVM.

Consult your operating system documentation on how to run the server in the background (for instance, using the & character at the end of the command line), on how to start the server automatically at boot time and stop it at shutdown time, and on how to restart the server periodically (for instance, using crontab).

Running the Server on Windows

On Windows you can also run the server using the `server.exe` application, which is located in the `bin` directory. For this to work the original structure of the ObjectDB directory must be preserved because `server.exe` tries to locate and load the objectdb.jar from the same directory in which it is started.

By default, `server.exe` starts the server using the following command:

```
> java -server -Xms32m -Xmx512m com.objectdb.Server start
```

When running `server.exe` you can specify arguments for the JVM as well as for the server (excluding the `start`, `stop` and the `restart` server commands). For example:

```
> server.exe -client -Xmx256m -port 6666
```

Explicitly specified arguments override defaults, so the above run uses the following command:

```
> java -client -Xms32m -Xmx256m com.objectdb.Server -port 6666 start
```

When running, the `server.exe` application is represented by an icon in the Windows Tray.
Right click the icon and use the context menu to manage the server (stop, restart and start), and to exit the server application.

5.3 JPA / JDO Class Enhancer

ObjectDB Enhancer is a post compilation tool that improves performance by modifying the byte code of compiled classes after compilation. Enhancement is mainly for user-defined persistable classes (entity classes, embeddable classes and mapped superclasses), and is usually optional.

There is one case, however, where enhancement is required. Non persistable classes that access directly (not through methods) persistent fields of enhanced classes must also be enhanced. It is a good practice (and actually required by JPA but not enforced by ObjectDB) to avoid accessing persistent fields of other classes directly. Rather, the accessor and mutator methods of the desired class should be used (e.g. by using the `get` and `set` methods). If you follow this practice only user defined persistable classes should need to be enhanced.

The enhancer silently ignores any specified class that does not need to be enhanced.

Enhancement improves efficiency in three ways:

- Enhanced code enables efficient tracking of persistent field modifications, avoiding the need for snapshot comparison of entities (as explained in the chapter 3). This is done by adding special code to enhanced classes that automatically notifies ObjectDB whenever a persistent field is modified.
- Enhanced code enables lazy loading of entity objects. With no enhancement, only persistent collection and map fields can be loaded lazily (by using proxy objects), but persistent fields that reference entity objects directly have to be loaded eagerly.
- Special optimized methods are added to enhanced classes as a replacement for using reflection. These optimized methods are much faster than using reflection.

Command Line Enhancement

ObjectDB Enhancer is a Java console application. It is contained in the `objectdb.jar` file.

You can run it from the command line as follows:

```
> java -cp objectdb.jar com.objectdb.Enhancer
```

If `objectdb.jar` is not in the current directory a path to it has to be specified.

Alternatively, you can run the Enhancer by using a shell script (`enhancer.bat` on Windows and `enhancer.sh` on Unix/Linux) from the ObjectDB `bin` directory. To use that script you have to edit the paths to the`objectdb.jar` file and to the JVM.

A usage message is displayed if no arguments are specified on the command line:

```
ObjectDB Enhancer [version 2.0]
Copyright (c) 2010, ObjectDB Software. All rights reserved.

Usage: java com.objectdb.Enhancer [ <options> | <class> | <filename> ] ...
 <class> - name of a class (without .class suffix) in the CLASSPATH
 <filename> - path to class or jar file(s), *? wildcards supported
 <options> include:
 -cp <dir>  :  path to input user classes
 -pu <name> :  persistence unit name
 -s         :  include sub directories in search
 -d <dir>   :  output path for enhanced classes
```

You can specify class files and jar files for enhancement explicitly or by using wildcards:

```
> java com.objectdb.Enhancer test/*.class Main.class pc.jar
```

If the `-s` option is specified, files in subdirectories are also searched and enhanced:

```
> java com.objectdb.Enhancer -s "*.class"
```

The `"*.class"` expression above is enclosed in quotes to prevent extraction by the shell.

The result output message lists the classes that have been enhanced:

```
[ObjectDB 2.0]
3 persistable types have been enhanced:
 test.MyEntity1
 test.MyEntity2
 test.MyEmbeddable
2 NON persistable types have been enhanced:
 Main
 test.Manager
```

You can also specify names of classes that can be located on the classpath using the syntax of import statements (e.g. `test.X` for a single class, `test.pc.*` for a package):

```
> java com.objectdb.Enhancer test.X test.pc.*
```

Use the `-pu` option with the name of a persistence unit to enhance all the managed classes that are defined in that persistence unit:

```
> java com.objectdb.Enhancer -pu my-pu
```

The `-cp` option can be used to specify an alternative classpath (the default is the classpath in which the Enhancer itself is running):

```
> java com.objectdb.Enhancer -cp src test.X test.pc.*
```

By default, classes are enhanced in place, overriding the original class and jar files. Use the `-d` option to redirect output to a different directory, thus keeping the original files unchanged:

```
> java com.objectdb.Enhancer -s "*.class" -d enhanced
```

Maven and ANT Enhancement

Enhancement can be integrated into the build process.

The following Maven build file defines a Java compiler plugin that includes enhancement:

```
<build>
  ...
  <plugins>
  ...
    <plugin>
      ...
      <dependencies>
        <dependency>
          <groupId>com.objectdb</groupId>
          <artifactId>objectdb</artifactId>
          <version>2.2.2</version>
          <scope>compile</scope>
        </dependency>
      </dependencies>

      <executions>
        <execution>
          <phase>compile</phase>
          <goals>
```

```
            <goal>java</goal>
          </goals>
        </execution>
      </executions>

      <configuration>
        <mainClass>com.objectdb.Enhancer</mainClass>

        <arguments>
          <argument>com.x.y.a.*</argument>
          <argument>com.x.y.b.*</argument>
        </arguments>
      </configuration>
   </plugin>
   ...
 </plugins>
 ...
</build>
```

Similarly, enhancement can be also be integrated into an ANT build script, as so:

```
<java classname="com.objectdb.Enhancer" fork="true"
    classpath="c:\objectdb\bin\objectdb.jar">
  <arg line="-s c:\my-project\classes\*.class"/>
</java>
```

Enhancement API

The ObjectDB Enhancer can also be invoked from Java code:

```
com.objectdb.Enhancer.enhance("test.pc.*,test.X");
```

The same arguments that can be specified on the command line can also be passed to the enhance method as a single string delimited by commas or spaces. In addition, a class loader for loading classes for enhancement can be specified as a second argument:

```
com.objectdb.Enhancer.enhance(
  "test.pc.*,test.X", text.X.class.getClassLoader());
```

The enhancement API and invocation of the Enhancer from Java code is useful, for instance, in implementing custom enhancement ANT tasks.

Load Time (Java Agent) Enhancement

Instead of enhancing classes during build, classes can be enhanced when they are loaded into the JVM by running the application with `objectdb.jar` as a Java agent. For example, if `objectdb.jar` is located at `c:\objectdb\bin`, the JVM can be started by:

```
> java -javaagent:c:\objectdb\bin\objectdb.jar MyApplication
```

If the JVM is run with ObjectDB Enhancer as a Java Agent, every loaded class is checked and automatically enhanced in memory (if applicable). Notice, however, that only classes which are marked as persistable by annotations (e.g. @Entity, @Embeddable) are enhanced by the Java Enhancer Agent. Therefore, when using this technique persistent fields may only be accessed directly from annotated persistable user classes.

Enhancement by a Java agent is very easy to use and convenient during development. For release, however, it is recommended to integrate the enhancement in the build process.

To use load time enhancement in web applications the web server or application server has to be run with the Java agent JVM argument.

Setting a Java Agent Enhancer in the IDE

In Eclipse JVM arguments can be set globally at:

`Window > Preferences > Java > Installed JREs > Edit > Default VM Arguments`

or for a specific run configuration, at:

`Run Configurations… > Arguments > VM arguments`

In NetBeans JVM arguments can be set at the project properties:

Right clicking the project > `Properties > Run > VM Options`

Automatic Java Agent Enhancer in JDK 6

Unless configured otherwise, ObjectDB tries to load the Enhancer as a Java Agent and enhance classes on the fly during load, even if a Java Agent is not specified explicitly.

This enhancement technique is inferior to the other techniques that are described above. First, currently it

only works on Sun JDK 6 (and not on JRE 6 for example). Second, classes that are loaded into the JVM before accessing ObjectDB cannot be enhanced, so a careful organization of the code is essential in order to make it work.

Therefore, specifying a Java Agent explicitly, as explained above should always be preferred.

5.4 Database Replication and Clustering

ObjectDB supports master-slave replication (cluster) since version 2.1. When replication (or clustering) is used, the same database is managed on multiple machines (nodes), possibly in different geographic locations. This could help in achieving high availability, fault tolerance and prompt disaster recovery.

In master-slave replication the master node manages the main (master) database, which supports Read / Write operations. The other (slave) nodes in the cluster manage identical copies of the same database, which are limited to READ operations. Any update to the master database is automatically propagated to the slave databases, keeping all the slave databases in the cluster synchronized with the master database.

Setting a Master Database

A master ObjectDB database is an ordinary database, which is managed on an ordinary ObjectDB database server. Any ObjectDB database on any server (but not in embedded mode) can function as a master database in a cluster. Recording has to be enabled, but no other preparations or settings are required.

Setting Slave Databases

Setting slave databases is very easy and only requires running an ObjectDB database server with appropriate `<replication>` elements in the configuration:

```
<server>
    <connection port="6001" max="100" />
    <data path="$objectdb/db-files" />
    <replication url="objectdb://localhost:6000/test.odb;user=a;password=a" />
</server>
```

The url attribute of the `<replication>` element defines a master database. As demonstrated above, a full url has to be specified including `user` and `password` attributes. The slave server uses these details to connect to the master server in order to listen to updates. The updates are automatically applied on the slave database, keeping it synchronized with the master database.

The same ObjectDB server can manage different types of databases, including master databases, slave database (by using one or more `<replication>` elements) and also databases that are not part of any cluster.

The replicated databases on the slave server are automatically generated under a special root directory, `$replication`, under the server data root directory. Starting a new replication of an existing master database requires copying the existing master database to the slave side. This is performed by ObjectDB automatically, but note that it could consume significant time if the existing database is large.

Connecting to the Database Cluster

The configuration above demonstrates a situation in which the master database is managed by a server on `localhost:6000` and the slave database is managed by a server on `localhost:6001`.

In this case, the master database can be accessed ordinarily as follows:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost:6000/test.odb;user=a;password=a");
```

A slightly different url is required in order to access the slave database:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost:6001//localhost:6000/test.odb;user=b;password=b");
```

The above url specifies the location of the slave server on port 6001 as well as the location of the master server on port 6000. Notice that the specified user and password attributes should represent a user on the slave server rather than on the master server.

Finally, a composite url can also be used:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost:6000/test.odb;user=a;password=a|" +
    "objectdb://localhost:6001//localhost:6000/test.odb;user=b;password=b"
);
```

A composite url contains two or more database urls separated by `'|'`. Usually only the first url (the master database in this example) is used. But when the first url becomes unavailable the connection will automatically switch (temporarily) to the next available url (i.e. in this example to the slave database) until the first url becomes available again.

5.5 Online Backup

An ObjectDB database can be backed up by simply copying or archiving the database file while the database is offline (i.e. when it is not open in an ObjectDB server and not in use by any application), since an ObjectDB database is stored as an ordinary file in the file system.

ObjectDB 2.1 introduces support of online backup, which enables backing up an ObjectDB database while it is in use. This is useful in applications that provide round the clock service (24/7/365) such as most web applications.

Starting Online Backup

The online backup can be started by executing a special query on an `EntityManager` (`em`) that represents the connection (local or remote) to the database:

```
em.createQuery("objectdb backup").getSingleResult();
```

The backup query string is always exactly `"objectdb backup"`.

The backup is created under the backup root directory, which by default is `$objectdb/backup`, i.e. a subdirectory of the ObjectDB home directory (and in client-server mode a subdirectory of the ObjectDB home directory on the server side).

A new subdirectory with a name that reflects the current date and time (e.g. 201012312359) is created under the backup root directory and the backup database file itself is created in that subdirectory with the name of the original database file.

For example, backing up a test.odb database file using the code above could generate a backup whose full path is c:\objectdb\backup\201012312359\test.odb (if `c:\objectdb` is the ObjectDB home directory).

Custom Backup Target

A custom backup root directory can be specified by setting the `target` parameter before executing the backup query:

```
Query backupQuery = em.createQuery("objectdb backup");
backupQuery.setParameter("target", new java.io.File("c:\backup"));
backupQuery.getSingleResult();
```

The code above, for instance, could create a backup at c:\backup\201012312359\test.odb regardless of the ObjectDB home directory location.

Notice that if the `target` argument is specified as a `java.io.File` instance and `em` represents a client-server connection then the backup file will be downloaded to the client and will be stored on the client machine.

Alternatively, the backup target could be specified as a `String` value:

```
Query backupQuery = em.createQuery("objectdb backup");
backupQuery.setParameter("target", "backup");
backupQuery.getSingleResult();
```

When a string is specified as a value for the target parameter it represents a path relative to the ObjectDB home directory, and in client-server mode, relative to the ObjectDB home directory on the server side.

For example, in client-server mode the code above could create the backup file on the server side at backup/201012312359/test.odb under the ObjectDB home directory.

Online Backup Thread

Executing the backup query starts the backup asynchronously. Therefore, the backup query returns after the backup is started but at that time the backup operation may still be in progress.

The backup query returns as a result a `Thread` instance that represents the backup run. This thread can be used, for example, for waiting until the backup is completed.

```
TypedQuery<Thread> backupQuery =
    em.createQuery("objectdb backup", Thread.class);
Thread backupThread = backupQuery.getSingleResult();
backupThread.join(); // Wait until the backup is completed.
// Do something with the backup (e.g. upload it to Amazon S3).
```

Notice that in client-server mode the returned `Thread` is a local thread on the client side that represents the real thread on the server side. Therefore, some operations (e.g. changing the thread priority) will have no real effect in client-server mode.

5.6 Database Doctor

The ObjectDB Doctor tool provides two related services:

- **Diagnosis and validation of an ObjectDB database file**
  Checks a given ObjectDB database file, verifies that it is healthy and valid, and if the file is not valid (it is corrupted), produces detailed diagnosis report of all the errors.
- **Repair of a corrupted ObjectDB database file**
  Repairs a corrupted ObjectDB database file by creating a new fresh database file and then copying all the recoverable data in the corrupted database file to the new database file.

Corrupted Database Files

Database files may be damaged and become corrupted due to various reasons:

- Hardware failure (e.g. a physical disk failure).
- Software failure (e.g. a bug of the Operating System, Java or ObjectDB).
- Copying a database file while it is open and in use.
- Network or I/O failure when copying, moving or transferring a database file.
- Transferring a database file over FTP in ASCII mode (BINARY mode should be used).
- Deleting an ObjectDB database recovery file or recording directory if it exists, or copying, moving or transferring an ObjectDB database file without its recovery file or recording directory.
- Power failure when the database is being updated - if recovery file is disabled.
- Using the database file simultaneously by two instances of the ObjectDB engine (not using one server process), thus bypassing ObjectDB internal file lock protection.
- Modifying the database file externally not through ObjectDB (e.g. by malicious software such as a computer virus).

Given all these causes it is clear that database files should be backed up regularly and often.
It is also recommended to validate production database files (or their backups) often by running ObjectDB Doctor's diagnosis regularly in order to identify potential problems early on.

Running ObjectDB Doctor Diagnosis

The ObjectDB Doctor tool is bundled in the `objectdb.jar` file.

It can be run from the command line:

```
> java -cp objectdb.jar com.objectdb.Doctor my.odb
```

If `objectdb.jar` is not in the current directory a path to it has to be specified.

The tool main class is `com.objectdb.Doctor` and the only command line argument for running a database diagnosis is the path to the database file (e.g. `my.odb` as shown above).

Diagnosis results are printed to the standard output.

Running ObjectDB Doctor Repair

Running the ObjectDB Doctor in repair mode requires specifying two command line arguments:

```
> java -cp objectdb.jar com.objectdb.Doctor old.odb new.odb
```

The first argument (`old.odb` above) is the path to the original (corrupted) database file.
The second argument (`new.odb` above) is the path to the new database file to be generated by the ObjectDB Doctor.

5.7 Database Transaction Replayer

ObjectDB can record its internal engine operations in special binary recording (journal) files. Recording is enabled by default and can be disabled in the configuration.

The ObjectDB Replayer tool can apply recorded database operations on a matching database backup (if available). This ability is useful for two different purposes:

- It enables recovery from a database failure by replaying the recorded operations.
- It enables reproducing problems during debugging by repeating a failure.

Backup & Recording Files

When recording is enabled, ObjectDB maintains for every database file a recording directory whose name is the name of the database file with the `odr` (ObjectDB Recording) suffix.

By default, the recording directory is generated in the directory that contains the database file. If the purpose of the recording is data durability it might be useful to keep the recording directory on a different physical device by setting the `path` attribute in the configuration.

The recording directory contains two types of files:

- Backup files - with names of the form `<transaction-id>.odb`

- Recording files - with names of the form `<transaction-id>.odr`

A backup file is an ordinary ObjectDB database file that reflects the state of the database at the end of a specific transaction. The ID of that transaction is used as the name of the file.

A recording file, with the same transaction ID in its name, contains database operations that have been recorded after that transaction.

Recorded operations can be replayed only if a proper backup file exists. Therefore, when recording is enabled and the required backup file does not exist, ObjectDB automatically creates a backup file as a copy of the existing ObjectDB database file when the database is opened. Preparation of the initial backup might be slow if the database is large.

Running the ObjectDB Replayer

The ObjectDB Replayer tool is bundled in the `objectdb.jar` file.

It can be run from the command line:

```
> java -cp objectdb.jar com.objectdb.Replayer my.odb
```

If `objectdb.jar` is not in the current directory a path to it has to be specified.

The tool's main class is `com.objectdb.Replayer` and the required argument is the path to the database file (e.g. `my.odb` as shown above). ObjectDB automatically locates the proper backup and recording files and tries to apply all the recorded operations. The resulting database file is also generated in the recording directory as `<transaction-id>.odb`, which specifies by its name the last executed transaction.

The Replayer can also be run up to a specified transaction, e.g.:

```
> java -cp objectdb.jar com.objectdb.Replayer my.odb 1000
```

When a transaction ID is specified as a second argument the Replayer applies recorded operations only until that specific transaction is reached.

If the above run succeeds, and all the operations until transaction 1000 are applied, the generated result file is expected to be `1000.odb`.

5.8 BIRT/ODA ObjectDB Driver

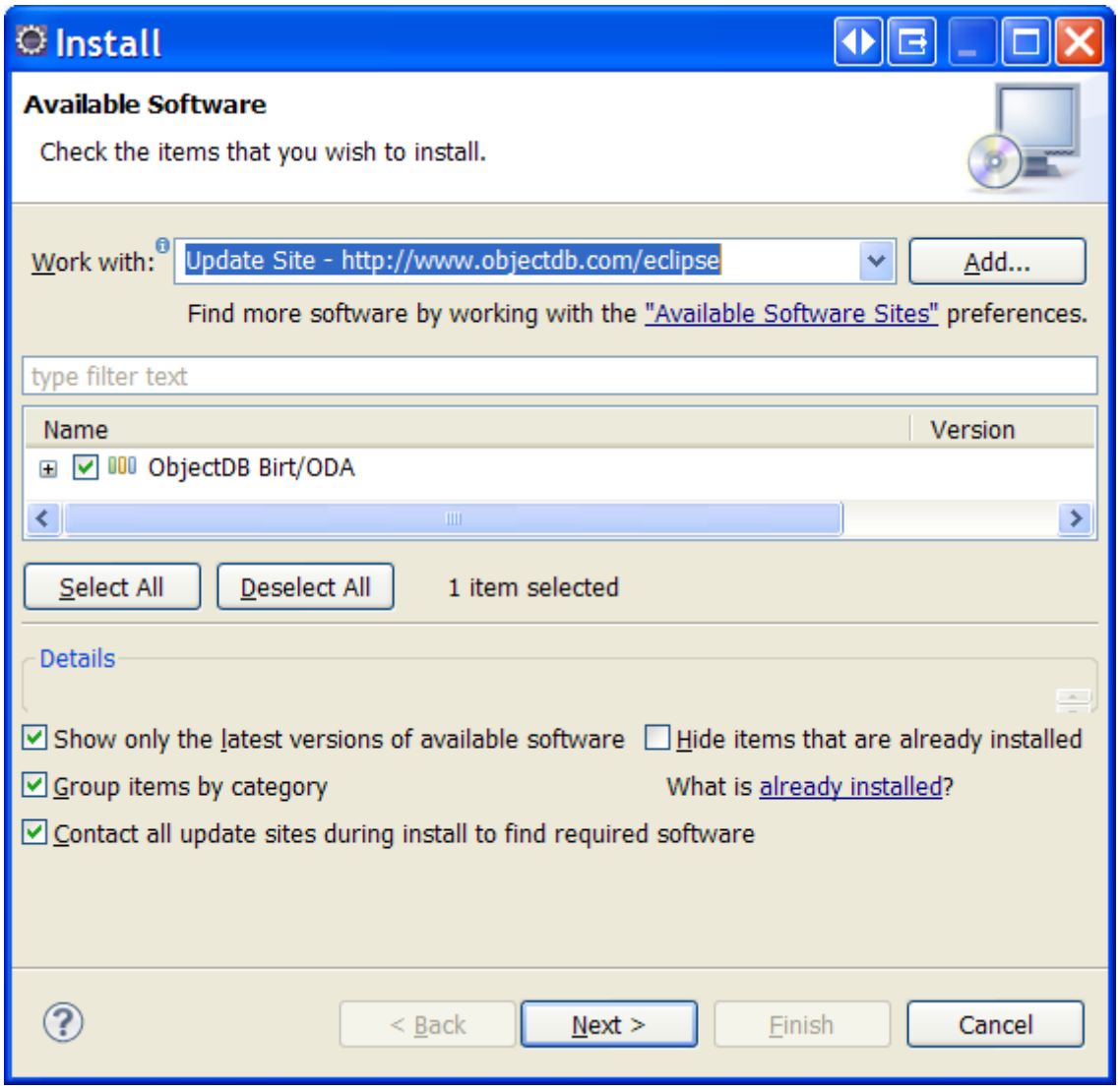
The ObjectDB BIRT/ODA driver is an extension of the open source Business Intelligence and Reporting Tools (BIRT) that adds support of ObjectDB as a data source and JPQL as a data set query language.

For step by step instructions on using BIRT with ObjectDB see the Report Generation with BIRT and JPA tutorial.

Driver Installation

The driver is available as an Eclipse for Java EE Developers extension. To install it:

- Open the [Install] dialog box by selecting **Help > Install New Software...**
- In the [Work with] field enter the ObjectDB update site url: **http://www.objectdb.com/eclipse** and press ENTER.
- Select the **ObjectDB Birt/ODA** feature.

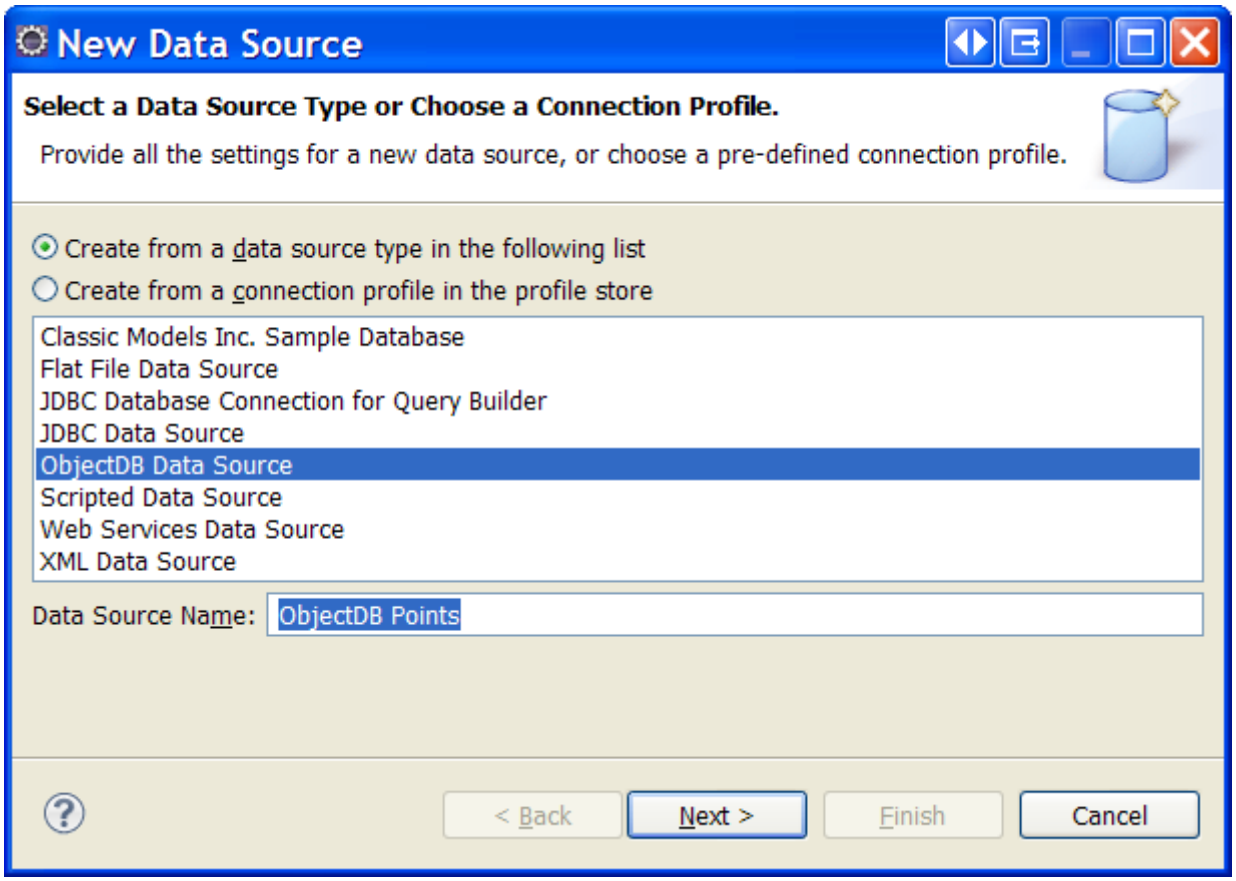


- Complete the installation by clicking **Next** twice, accepting the license agreement, clicking **Finish**, and finally approving the installation and restarting the IDE.

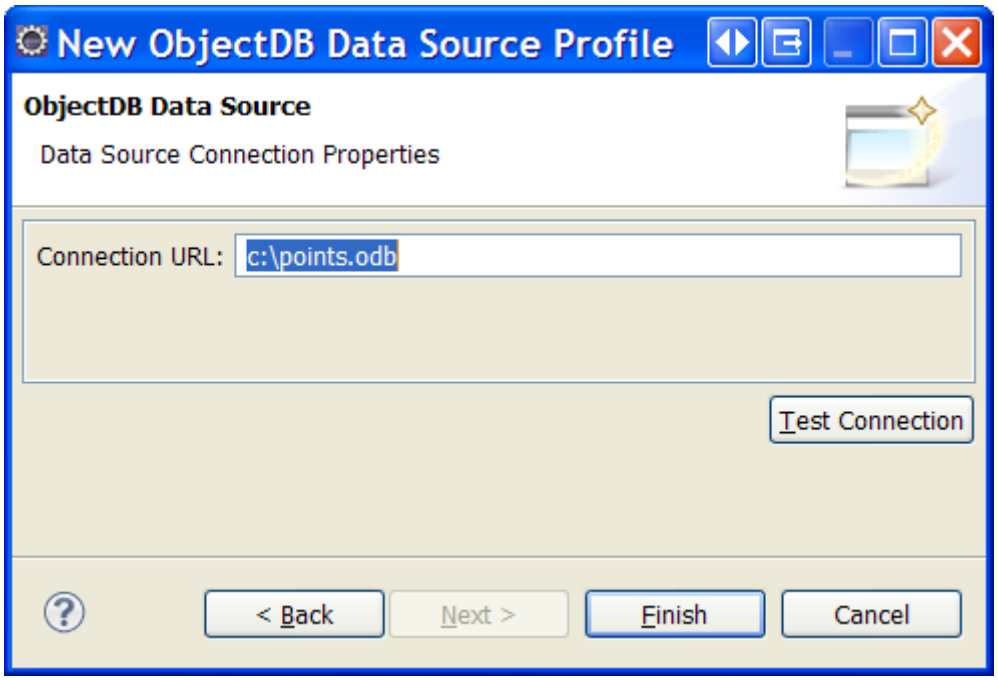
ObjectDB/JPA Data Source

To create an ObjectDB data source:

- Open the [New Data Source] dialog box by right clicking the **Data Sources** node in the [Data Explorer] window and selecting **New Data Source**.
- Select **ObjectDB Data Source** from the list of available data source types.
- Enter data source name (e.g. **ObjectDB Points**) and click **Next**.



- Specify an ObjectDB connection url - either embedded (e.g. **c:\points.odb**) or client-server (e.g. **objectdb://localhost/points.odb;user=admin;password=admin**).

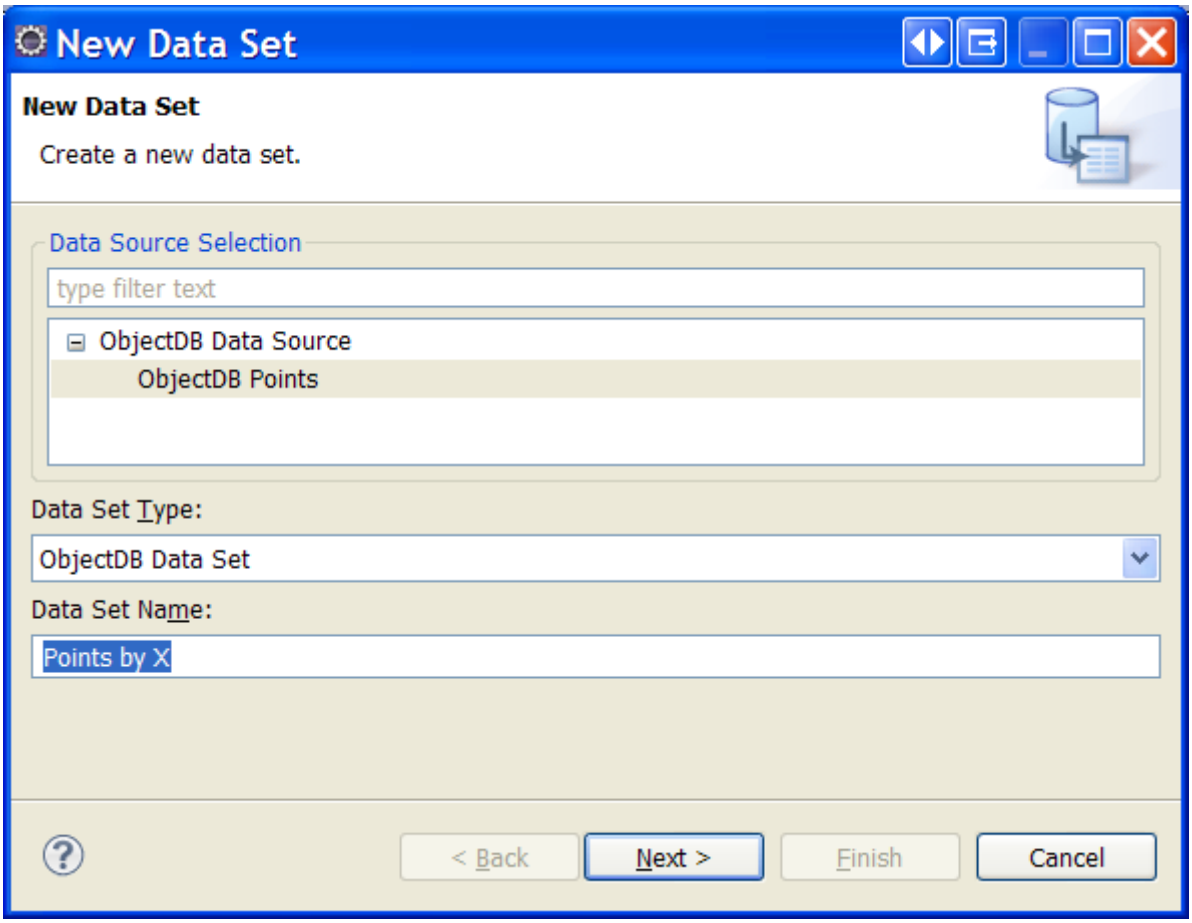


- Click the **Finish** button to complete the creation of the ObjectDB data source.

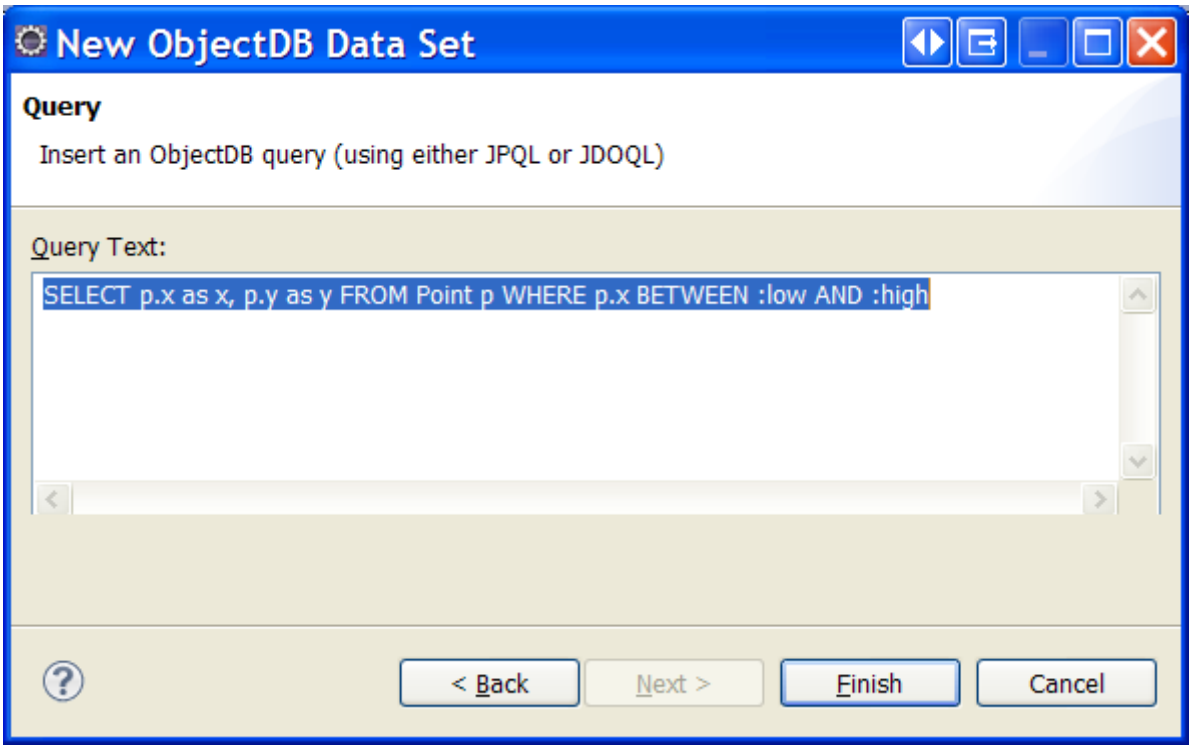
Data Sets and JPQL

To create the data set:

- Open the [New Data Set] dialog box by right clicking the **Data Sets** node in the [Data Explorer] window and selecting **New Data Set**.
- Select an ObjectDB data source (e.g. **ObjectDB Points**).
- Enter a data set name (e.g. **Points by X**) and click **Next**.



- Entry a JPQL or a JDOQL query and click **Finish**.



See the Report Generation with BIRT and JPA tutorial for more detailed instructions.

Chapter 6 - Configuration

The ObjectDB configuration file contains one <objectdb> root element with seven subelements:

```
<objectdb>
  <general> ... </general>
  <database> ... </database>
  <entities>  ... </entities>
  <schema> ... </schema>
  <server> ... </server>
  <users> ... </users>
  <ssl> ... </ssl>
</objectdb>
```

Each one of these seven configuration elements is explained in a separate section:

- General Settings and Logging
- Database Management Settings
- Entity Management Settings
- Schema Update
- Server Configuration
- Server User List
- SSL Configuration

This page explains how ObjectDB configuration works in general.

The Configuration Path

By default, the configuration file is loaded from `$objectdb/objectdb.conf` where `$objectdb` represents the ObjectDB home directory.

ObjectDB Home ($objectdb)

The value of `$objectdb` (the ObjectDB home directory) is derived from the location of the `objectdb.jar` file. It is defined as the path to the directory in which `objectdb.jar` is located,
with one exception - if the name of that directory is `bin`, `lib` or `build` the parent directory is considered to be the ObjectDB home directory (`$objectdb`).

As a result, `$objectdb` is also the installation directory of ObjectDB since `objectdb.jar` is located in the `bin` directory under the installation directory. Notice, however, that moving `objectdb.jar` to another

location changes the value of $objectdb. For example, in a web application, in which objectdb.jar is located in WEB-INF/lib, the ObjectDB home directory ($objectdb) is WEB-INF.

You can also define $objectdb explicitly by setting the "objectdb.home" system property:

```
System.setProperty("objectdb.home", "/odb"); // new $objectdb
```

As with any other system property it can also be set as an argument to the JVM:

```
> java "-Dobjectdb.home=/odb" ...
```

The Configuration File

As noted above, by default the configuration file is loaded from $objectdb/objectdb.conf.

You can specify an alternative path by setting the "objectdb.conf" system property:

```
System.setProperty("objectdb.conf", "/my/objectdb.conf");
```

It can also be set as an argument to the JVM:

```
> java "-Dobjectdb.conf=/my/objectdb.conf" ...
```

If a configuration file is not found default values are used.

General Configuration Considerations

The following rules apply to all the relevant configuration elements and attributes:

- $objectdb, representing the ObjectDB home directory, and $temp, representing the system default temporary path, can be used in any path attribute value in the configuration file.
- The mb and kb suffixes, representing megabytes and kilobytes (respectively), can be used in any size value attribute in the configuration file.
- Appropriate file system permissions have to be set for all the paths that are specified in the configuration file (for the process that runs ObjectDB).

6.1 General Settings and Logging

The <general> configuration element specifies ObjectDB settings that are relevant to both the server side and the client side.

The default configuration file contains the following <general> element:

```
<general>
  <temp path="$temp/ObjectDB" threshold="64mb" />
  <network inactivity-timeout="0" />
  <url-history size="50" user="true" password="true" />
  <log path="$objectdb/log/" max="8mb" stdout="false" stderr="false" />
  <log-archive path="$objectdb/log/archive/" retain="90" />
  <logger name="*" level="info" />
</general>
```

The <temp> element

```
  <temp path="$temp/ObjectDB" threshold="64mb" />
```

To meet memory constraints ObjectDB can use temporary files in processing of large data, such as query results that contain millions of objects.

The <temp> element specifies temporary file settings:

- The path attribute specifies a directory in which the temporary files are generated. The $temp prefix can be used to represent the system default temporary path, as demonstrated above.
- Using RAM is much faster than using temporary files. Therefore, temporary files are only used for data that exceeds a limit size that is specified by the threshold attribute. The mb and kb suffixes represent megabytes and kilobytes (respectively).

The <network> element

```
  <network inactivity-timeout="0" />
```

The <network> element has one attribute, inactivity-timeout, which specifies when network sockets become obsolete as a result of inactivity. The value is the timeout in seconds, where 0 indicates never (no inactivity timeout).

The inactivity timeout (when > 0) is applied on both the server side and the client side, when using client-server mode, and has no effect in embedded mode.

Specifying an inactivity timeout may solve firewall related issues. In general, if the firewall enforces its own inactivity timeout on sockets a more restrictive inactivity timeout has to be specified for ObjectDB to avoid using sockets that are expired by the firewall.

The <url-history> element

```
<url-history size="50" user="true" password="true" />
```

ObjectDB manages a list of the recently accessed database urls for use by the Explorer.

- The `size` attribute specifies the maximum size of this list. This feature can be disabled by specifying `0` as the maximum list size.
- The `user` attribute specifies if user names should be saved with urls (in client server mode).
- The `password` attribute specifies if passwords should also be saved with urls.

Saving username and password with the url makes accessing recently used databases in the Explorer easier.

The <log> element

```
<log path="$objectdb/log/" max="8mb" stdout="false" stderr="false" />
```

General logging settings are specified in the `<log>` element:

- The `path` attribute specifies a directory in which the log files are generated. The $objectdb prefix can be used to represent the ObjectDB installation directory, as demonstrated above.
- Every day a new log file is generated with the name `odb<yyyymmdd>.log` (where `<yyyymmdd>` represents the date). A new log file is also generated when the log file exceeds the maximum size, which is specified by the `max` attribute.
- The `stdout` and `stderr` attributes specify if log messages should also be written to the standard output and the standard error (respectively) in addition to writing to the log file.

The <log-archive> element

```
<log-archive path="$objectdb/log/archive/" retain="90" />
```

Old log files are moved to an archive directory.

The <log-archive> element specifies the logging archive settings:

- The path attribute specifies an archive directory.
- The retain attribute specifies how many days to keep the archived log files. After that period an archived log file is automatically deleted.

The <logger> elements

```
  <logger name="*" level="info" />
```

<logger> elements specify logging levels. The "*" logger, which can be shown in the default configuration above, represents the entire ObjectDB system.

Additional <logger> elements can be added to override the default logging level for a specific ObjectDB subsystem. The names of the subsystem loggers are currently undocumented and can change at any time without notice.

The supported logging levels are:

- "fatal"
- "error"
- "warning"
- "info"
- "trace"
- "debug"

6.2 Database Management Settings

The <database> configuration element specifies back end (database engine) settings which are relevant on the server side and in embedded mode.

The default configuration file contains the following <database> element:

```
<database>
  <size initial="256kb" resize="256kb" page="2kb" />
  <recovery enabled="true" sync="false" path="." max="128mb" />
  <recording enabled="false" sync="false" path="." mode="write" />
  <locking version-check="true" />
```

```
  <processing cache="64mb" max-threads="10" synchronized="false" />
  <query-cache results="32mb" programs="500" />
  <extensions drop="temp,tmp" />
</database>
```

The <size> element

```
  <size initial="256kb" resize="256kb" page="2kb" />
```

The `<size>` element specifies the database file and page size settings:

- The `initial` attribute specifies an initial size of every new database file. The `resize` attribute specifies the size by which to extend the database file when additional space is needed. Small initial size and resize values save space. Larger values can improve performance by reducing file fragmentation (many resize operations might cause fragmentation of the database file).
- The page attribute specifies the size of a page in a database file. The default of 2KB is appropriate for most applications.

The <recovery> element

```
  <recovery enabled="true" sync="false" path="." max="128mb" />
```

When enabled, a recovery file is created by ObjectDB when a database is opened and deleted by ObjectDB when the database is closed. The name of the recovery file is based on the name of the database file with $ added at the end. Every transaction commit is first written to the recovery file and then to the database. This way, if the system crashes during a write to the database, the recovery file can be used to fix the database. Recovery from failure is automatically applied by ObjectDB when a database is opened and a recovery file exists, indicating that it has not been closed properly. Moving or copying a database file that has not been closed properly without its recovery file may corrupt the database.

The `<recovery>` element specifies the recovery file settings:

- The `enabled` attribute (whose value is `"true"` or `"false"`) specifies if recovery file is used.
- The `sync` attribute (whose value is `"true"` or `"false"`) specifies if physical writing is required before `commit` returns. `sync=false` is much faster in writing data to a database, but `true` might be safer in production.
- By default, the recovery file is generated in the directory of the database file, but any other alternative path can be specified by the `path` attribute. Using separate storage devices (e.g. disks) for the recovery

file and the database file can improve performance.
- The max attribute is a hint that specifies the space that should be available for the recovery file (ObjectDB might use more space when necessary).

The <recording> element

```
<recording enabled="false" sync="false" path="." mode="all" />
```

Database engine operations can be recorded in files and later replayed using the ObjectDB Replayer tool. Recording provides an alternative (which is sometimes more efficient) to Replayer tool. Recording might also be useful for backup purposes and for debugging (by providing the ability to reproduce problems by replay)

The <recording> element specifies the recording settings:

- The enabled attribute (whose value is "true" or "false") specifies if recording is used.
- The sync attribute (whose value is "true" or "false") specifies whether physical writing is required for every recorded operation before returning to the caller.
- By default, a recording subdirectory is generated in the directory of the database file, but any other alternative path can be specified by the path attribute.
- The mode attribute (whose value is "all" or "write") specifies which operations should be recorded. For backup purposes only "write" operations (which modify the database) have to be recorded. For debugging of query failure it might be necessary to record "all" operations in order to reproduce the problem. Naturally, the recording operation is slower and the recording files are much larger when "all" is used.

The <locking> element

```
<locking version-check="true" />
```

The version-check attribute of the <locking> element specifies if optimistic locking is enabled. Optimistic locking is completely automatic and enabled by default in ObjectDB, regardless if a version field (which is required by some ORM JPA providers) is defined in the entity class or not.
It can be disabled by setting the version-check attribute to false.

The <processing> element

```
<processing cache="64mb" max-threads="10" />
```

The <processing> element specifies miscellaneous database engine settings:

- The cache attribute is a hint that specifies the the amount of memory that is used for caching pages of the database file.
- The max-threads attribute specifies the maximum number of concurrent threads that can be served by the database engine simultaneously. When the specified maximum is reached - new requests are pending until previous requests are completed. The optimal number is usually larger than the number of available CPU cores, but not too large, to avoid thread competition that leads to poor performance.

The <query-cache> element

```
<query-cache results="32mb" programs="500" />
```

The <query-cache> element specifies settings of the two cache mechanisms that ObjectDB manages for queries:

- The results attribute specifies the size of the query result cache. Caching results is very useful for recurring queries with identical arguments. As long as the relevant data in the database is unchanged cached results can be returned instead of running queries.
- The programs attribute specifies how many compiled query programs should be cached. Cached query programs may eliminate the need to compile queries again but the queries still have to be executed, so cached programs are less efficient than cached results. However, cached query programs can also be used for recurring queries with different arguments and are not affected by most database updates (except schema updates).

The <extensions> element

```
<extensions drop="temp,tmp" />
```

The drop attribute of the <extensions> element specifies a list of file name extensions that can be used for temporary databases (usually in tests). The content of these temporary databases is deleted when using the drop url connection parameter.

6.3 Entity Management Settings

The <entities> configuration element specifies front end settings that are relevant on the client side and in embedded mode.

The default configuration file contains the following <entities> element:

```
<entities>
  <enhancement agent="true" reflection="warning" />
  <cache ref="weak" level2="0mb" />
  <persist serialization="false" />
  <cascade-persist always="auto" on-persist="false" on-commit="true" />
  <dirty-tracking arrays="false" />
</entities>
```

The <enhancement> element

```
  <enhancement agent="true" reflection="warning" />
```

The <enhancement> element specifies enhancement related settings:

- The agent attribute (whose value is "true" or "false") specifies whether the Enhancer Agent should be loaded to enhance persistable types on the fly, even if it is not specified explicitly at the command line. This is currently supported only for JDK 6.0 (not JRE) or above.
- The reflection attribute specifies how non enhanced classes are handled. ObjectDB can manage non enhanced classes by using reflection at the cost of performance. The possible values of the reflection attribute represent different policies:
  - "error" - all persistable classes must be enhanced - otherwise an exception is thrown.
  - "warning" - a warning is logged for every non enhanced class.
  - "ignore" - reflection is used for non enhanced classes - with no error or warning.
  - "force" - reflection is used even for enhanced classes (for troubleshooting).

The <cache> element

```
  <cache ref="weak" level2="0mb" />
```

The <cache> element specifies settings of the two cache mechanisms for entities:

- The `ref` attribute specifies the reference type for holding non dirty entities in the persistence context of the `EntityManager` (which serves as a first level cache). The valid values are `"weak"`, `"soft"` and `"strong"`. Modified entities are always held by strong references in the persistence context (until `commit` or `flush`), regardless of this setting.
- The `level2` attribute specifies the size of the shared level 2 cache that is managed by the `EntityManagerFactory` and shared by all its `EntityManager` instances. The level 2 cache is disabled by specifying `0` or `0mb`.

The <persist> element

```
<persist serialization="false" />
```

The `serialization` attribute of the `<persist>` element (whose value is `"true"` or `"false"`) specifies if serialization should be used as a fallback persisting method for instances of serializable types that are non persistable otherwise (e.g. a user defined class, which is not an entity class, mapped super class or embeddable class).

The <cascade-persist> element

```
<cascade-persist always="auto" on-persist="false" on-commit="true" />
```

The `<cascade-persist>` element specifies global settings for cascading persist operations:

- The always attribute (whose value is `"true"`, `"false"` or `"auto"`) specifies if persist operations should always be cascaded for every entity, regardless local cascade settings.
  The `"auto"` value functions as `"true"` when using JDO and as `"false"` when using JPA.
- The `on-persist` attribute specifies whether cascading (as a result of either global or local setting) should be applied during persist.
- The `on-commit` attribute specifies whether cascading (as a result of either global or local setting) should be applied during commit and flush.

Note: Both JPA and JDO require cascading the persist operation twice, first during persist and later on commit or flush. Usually, commit time only cascade (which is more efficient than double cascade) is sufficient.

The <dirty-tracking> element

```
  <dirty-tracking arrays="false" />
```

The `arrays` attribute of the `<dirty-tracking>` element specifies if modifications to array cells should be tracked automatically in enhanced classes. See the Updating Entities section in chapter 3 for more details.

6.4 Schema Update

The `<schema>` configuration element supports renaming packages, classes and fields in ObjectDB databases as a complementary operation to renaming or moving these elements in the IDE during source code refactoring. Only these schema changes are specified in the configuration file. As explained in chapter 2, other schema changes are handled by ObjectDB automatically.

Note: Extreme caution is required when persistable classes are renamed or moved to another package. Running the application with persistable classes that have been renamed or moved in the IDE, with no matching schema configuration - will create new, separate persistable classes with no instances. Therefore, you should backup your database files before renaming or moving persistable classes and you must verify that after such changes the application is run only with the configuration that matches these changes exactly.

The default configuration file contains an empty `<schema>` element. If the `<schema>` element is not empty ObjectDB tries to apply the specified schema updates every time a database is opened. When using client-server mode the `<schema>` instructions should usually be located on the client side where the up to date classes are located.

The following `<schema>` element demonstrates the supported schema update abilities:

```
<schema>
  <package name="com.example.old1" new-name="com.example.new1" />
  <package name="com.example.old2" new-name="com.example.new2">
    <class name="A" new-name="NewA" />
    <class name="B">
      <field name="f1" new-name="newF1" />
      <field name="f2" new-name="newF2" />
    </class>
  </package>
  <package name="com.example.old3">
    <class name="C" new-name="NewC" >
      <field name="f3" new-name="newF3" />
    </class>
    <class name="C$E" new-name="NewC$E" />
```

```
  </package>
</schema>
```

The hierarchy, as demonstrated above, is strict:

- <package> elements are always direct child elements of the <schema> element.
- <class> elements are always direct child elements of <package> elements.
- <field> elements are always direct child elements of <class> elements.

The <package> elements

```
<package name="com.example.old1" new-name="com.example.new1" />
<package name="com.example.old2" new-name="com.example.new2">
   ...
</package>
<package name="com.example.old3">
  ...
</package>
```

A <package> element has two roles:

- If the optional new-name attribute is specified the package name is changed from the original name, which is specified by the required name attribute, to the new name. All the classes in that package are moved to the new package name.
- In addition, whether or not a new-name attribute is specified a <package> element serves as a container of <class> subelements for renaming classes and fields in that package.

The <package> elements above specify renaming of the com.example.old1 and com.example.old2 packages. The com.example.old3 package is not renamed, but rename operations are specified for some of its classes.

The <class> elements

```
  <class name="A" new-name="NewA" />
  <class name="B">
    ...
  </class>
```

```
    <class name="C" new-name="NewC" >
      ...
    </class>
    <class name="C$E" new-name="NewC$E" />
```

A <class> element has two roles:

- If the optional new-name attribute is specified the class name is changed from the original name, which is specified by the required name attribute, to the new name. The value of the name attribute must be unqualified (with no package name) because the package name is already specified in the containing <package> element. The value of the new-name attribute can be either qualified or unqualified. If it is unqualified (no package name) the new-name value of the containing <package> element is used, if it exists, or if no new-name is specified in the <package> element the name value of the <package> element is used.
- In addition, whether or not a new-name attribute is specified a <class> element serves as a container of <field> subelements for renaming fields in that class.

The <class> elements above specify renaming of the A, C and C.E (which has to be written as C$E) classes. Class B is not renamed but rename operations are specified for some of its fields.

The <field> elements

```
      <field name="f1" new-name="newF1" />
      <field name="f2" new-name="newF2" />

      <field name="f3" new-name="newF3" />
```

The <field> element specifies renaming a persistent field (or a property). Both attributes, the name, which specifies the old name, and new-name, which specifies the new name, are required.

6.5 Server Configuration

The <server> configuration element specifies settings for running an ObjectDB Server.

The server is affected also by other elements in the configuration file, particularly the <users> and the <ssl> configuration elements.

The default configuration file contains the following <server> element:

```
<server>
  <connection port="6136" max="100" />
  <data path="$objectdb/db-files" />

</server>
```

The <connection> element

```
  <connection port="6136" max="100" />
```

The <connection> element specifies how clients can connect to the server:

- The port attribute specifies a TPC port on which the server is listening for new connections. Usually the default port 6136 should be specified. If another port is specified it also has to be specified by clients in the url connection string when connecting to the database (as explained in the JPA Overview section in chapter 3).
- The max attribute specifies the maximum number of simultaneous connections that are accepted by the server. A request for a connection that exceeds the maximum is blocked until another open connection is closed.

The <data> element

```
  <data path="$objectdb/db-files" />
```

The <data> element has one attribute, path, which specifies the location of ObjectDB databases that the server manages. The $objectdb prefix, if specified (as demonstrated above), represents the ObjectDB home directory.

The data path of an ObjectDB server is similar to the document root directory of a web server. Every database file in the data directory and in its subdirectories can be accessed by the server. Appropriate file system permissions have to be set on the data directory, its subdirectories and database files, to enable operations of the server process.

When connecting to the server the path that is specified in the url connection is resolved relative to the data path. For example, "objectdb://localhost/my/db.odb" refers to a database file db.odb in a subdirectory 'my' of the data directory.

The <replication> elements

```
<replication url="objectdb://localhost/test.odb;user=admin;password=admin" />
```

The <replication> elements are optional elements that define replication of ObjectDB databases in a master-slave cluster. A <replication> element is only required on the slave server to define the replicated slave database. More details on replication are provided on the Database Replication and Clustering page.

6.6 Server User List

The <users> configuration element lists the users that are allowed to access the ObjectDB Server and specifies their specific settings (username, password, permissions, quota).

The default configuration file contains the following <users> element:

```
<users>
  <user username="admin" password="admin" ip="127.0.0.1" admin="true">
    <dir path="/" permissions="access,modify,create,delete" />
  </user>
  <user username="$default" password="$$$###">
    <dir path="/$user/" permissions="access|modify|create|delete">
      <quota directories="5" files="20" disk-space="5mb" />
    </dir>
  </user>
  <user username="user1" password="user1" />
</users>
```

The <user> elements

```
  <user username="admin" password="admin" ip="127.0.0.1" admin="true">
    ...
  </user>

  <user username="$default" password="$$$###">
    ...
  </user>
```

```
<user username="user1" password="user1" />
```

Every user is represented by a single <user> element:

- The required username and password attributes specify a username and a password that have to be provided when the user connects to the server.
- The optional ip attribute, if specified, restricts the user to connect to the server only from the specified IP addresses. For instance, "127.0.0.1" (which represents the local machine), as shown above, restricts the user to the machine on which the server is running.
  Multiple IP addresses can also be specified in a comma separated list and using a hyphen (-) to indicate a range. For example, a value "192.18.0.0-192.18.194.255,127.0.0.1" allows connecting from any IP address in the range of 192.18.0.0 to 192.18.194.255, as well as from 127.0.0.1.
- The admin attribute (whose value is "true" or "false") specifies if the user is a superuser. A superuser is authorized to manage server settings using the ObjectDB Explorer.

A value of "$default" for the username attribute indicates a virtual master user definition. All the settings of that master definition are automatically inherited by all the other user definitions but the master user itself cannot be used to connect to the database.

The <dir> element

```
<dir path="/" permissions="access,modify,create,delete" />

<dir path="/$user/" permissions="access|modify|create|delete">
  <quota directories="5" files="20" disk-space="5mb" />
</dir>
```

Every <user> element may contain one or more <dir> subelements indicating which paths under the server data directory the user is allowed to access:

- The required path attribute specifies a directory path relative to the root data directory. Permission to access a directory always includes the permission to access the whole tree of subdirectories under that directory. Therefore, path "/" indicates permission to access any directory in the data directory.
  $user represents the user's username and if specified for the master ("$default") it is interpreted by every concrete user definition as the real username of that user. This way, it is easy to allocate a private directory for every user.
- The required permissions attribute specifies which database file permissions are granted.
  The comma separated string value may contain the following permissions:

- `access` - permission to open a database for read.
- `modify` - permission to modify the content of a database.
- `create` - permission to create new subdirectories and database files.
- `delete` - permission to delete subdirectories and database files.

If no database file permissions are specified the user is still allowed to view the directory content (using the Explorer) but cannot open database files or modify anything.

The <quota> element

```
<quota directories="5" files="20" disk-space="5mb" />
```

Every `<dir>` element may contain one optional `<quota>` subelement, specifying restrictions on the directory content:

- The `directories` attribute specifies how many subdirectories are allowed under that directory (nested subdirectories are also allowed).
- The `files` attribute specifies how many database files the directory may contain.
- The `disk-space` attribute specifies maximum disk space for all the files in that directory.

6.7 SSL Configuration

The `<ssl>` configuration element specifies Secure Sockets Layer (SSL) settings for secure communication in client-server mode, for both the client side and the server side.

The default configuration file contains the following `<ssl>` element:

```
<ssl enabled="false">
  <server-keystore path="$objectdb/ssl/server-kstore" password="pwd" />
  <client-truststore path="$objectdb/ssl/client-tstore" password="pwd" />
</ssl>
```

The `enabled` attribute of the `ssl` element (whose value is `"true"` or `"false"`) specifies if SSL is used. As shown above, SSL is disabled by default. It could be enabled when accessing remote ObjectDB databases over an insecure network such as the Internet.

SSL Keystore and Truststore Files

To use SSL you have to generate at least two files:

- A **Keystore** file that functions as a unique signature of your server. This file contains general details (such as a company name), an RSA private key and its corresponding public key (the SSL protocol is based on the RSA algorithm).
- A **Truststore** file that functions as a certificate that enables the client to validate the server signature. This file is generated from the Keystore file by omitting the private key (it still contains the general information and the public key).

You can generate these files using the JDK **keytool** utility:

> http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html

Using these Keystore and Trustore files a client can verify during SSL handshaking that it is connected to the real server and not to another server in the way that is pretending to be the real server (what is known as "a man in the middle attack"). The server, on the other hand, might be less selective and allow connections from any machine as long as a valid username and password are provided. If an authentication of the client machine by the server is also required a Keystore file (which might be different from the server Keystore) has to be installed on the client machine and its corresponding Trustore file has to be installed on the server machine.

Setting the Configuration

```
<ssl enabled="true">
  <server-keystore path="$objectdb/ssl/server-kstore" password="pwd" />
  <server-truststore path="$objectdb/ssl/server-tstore" password="pwd" />
  <client-keystore path="$objectdb/ssl/client-kstore" password="pwd" />
  <client-truststore path="$objectdb/ssl/client-tstore" password="pwd" />
</ssl>
```

To use SSL the enabled attribute of the ssl element has to be set to true.

Every keystore / truststore file is represented by a separate child element with two required attributes: `path`, which specifies the path to the file, and `password`, which specifies a password that is needed in order to use the file.

Usually only the `server-keystore` element (for the server) and the `client-truststore` element (for the client) are needed (as shown above).

The other two elements, `client-keystore` and `server-truststore`, are needed only when the client is also signed (as explained above).