ObjectDB 2.0.1 Developer's Guide

Copyright @ 2010 by ObjectDB Software (http://www.objectdb.com)

Last updated on October 11, 2010.

Table of Contents

Preface 4
Chapter 1 - Quick Tour 5
1.1 Entity Class 5
1.2 Database Connection 7
1.3 CRUD Operations 9
1.4 What is Next? 12
Chapter 2 - JPA Entity Classes 13
2.1 Persistable Types 13
2.2 Entity Fields 19
2.3 Primary Key 24
2.4 Generated Values 27
2.5 Index Definition 30
2.6 Schema Evolution 34
2.7 Persistence Unit 36
Chapter 3 - Using JPA 39
3.1 Main Interfaces 39
3.2 Working with Entities 42
3.3 CRUD Operations 44
3.3.1 Storing Entities 45
3.3.2 Retrieving Entities 48
3.3.3 Updating Entities 52
3.3.4 Deleting Entities 54
3.4 Advanced Topics 56
3.4.1 Detached Entities 56
3.4.2 Lock Management 58
3.4.3 Lifecycle Events 61
Chapter 4 - JPA Queries and JPQL 65
4.1 Query API 65
4.1.1 Running Queries 66
4.1.2 Query Parameters 68
4.1.3 Setting & Tuning 71
4.1.4 Named Queries 73
4.2 Query Structure 75
4.2.1 JPQL SELECT 77
4.2.2 JPQL FROM 80
4.2.3 JPQL WHERE 84
4.2.4 JPQL GROUP BY 85

4.2.5 JPQL ORDER BY 88
4.3 Query Expressions 90
4.3.1 JPQL Literals 91
4.3.2 Objects In JPQL 94
4.3.3 Numbers in JPQL 95
4.3.4 Strings in JPQL 97
4.3.5 Collections in JPQL 99
4.3.6 Comparison Operators 100
4.3.7 Logical Operators 103
Chapter 5 - Object Database Tools 106
5.1 ObjectDB Explorer 106
5.2 ObjectDB Server 107
5.3 ObjectDB Enhancer 110
5.4 ObjectDB Doctor 114
5.5 ObjectDB Replayer 116
Chapter 6 - Configuration 118
6.1 General and Logging 120
6.2 Database Management 122
6.3 Entity Management 125
6.4 Schema Update 126
6.5 Server Configuration 129
6.6 Server User List 130
6.7 SSL Configuration 132

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 and JPQL

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

- Chapter 5 - Object Database Tools

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:

- Defining a JPA Entity Class
- Obtaining a JPA Database Connection
- CRUD Database Operations with JPA
- What is Next?

To run the sample program of this chapter in your IDE - you can follow a tutorial:

- 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. Giving 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 an ordinary POJO (Plain Old Java Object) class, which is marked as having the ability to manage and represent objects in the database. That might remind serializable classes, which are marked as having the ability to be used for serialization.

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 "(" + x + ',' + 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) and the values of these fields are stored in the database when an entity object is persisted.

The JPA Entity Classes manual pages provide 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 is a 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. ObjectDB supports all the JPA primary key variants, including composite primary keys and automatic sequential value generation. That is a very powerful and unique feature of ObjectDB that is absent from other object oriented databases. To let you enjoy the best of both words - ObjectDB supports also the traditional implicit object oriented database IDs, so explicit definition of a JPA primary key is optional.

1.2 Obtaining a JPA Database Connection

A connection to a database is represented in JPA by the `EntityManager` interface. In order to manipulate an ObjectDB database we need an `EntityManager` instance. Operations that modify database content require also 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 a 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 use a direct database file path as a replacement:

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

The `createEntityManagerFactory` static method expects a persistent 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 as a database url rather than as 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 that path - ObjectDB will try to create a new empty object database file at that path.

At the end of use, the database file has to be closed by closing the `EntityManagerFactory`:

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

Obtaining an EntityManager

An `EntityManager` instance can easily be obtained from an `EntityManagerFactory`:

```
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`. Usually in a multithreading application, every thread uses its own `EntityManager` instance, where one `EntityManagerFactory` instance (that represents the database) is shared by the entire application.

The connection to the database is closed be closing the `EntityManager`:

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

Closing an `EntityManager` does not close the database file. The `EntityManager` object itself cannot be reused after close, but the owner `EntityManagerFactory` may preserve its 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.

Giving 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 the 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 committed to the database when the transaction is being committed:

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

The next section explains in more details 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 a database (such as storing new objects in the database) require an active transaction. Every `Point` object is first constructed as an ordinary Java object. It 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. More details on persisting objects in the database are provided in the Storing Entities section of this manual.

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 an SQL like syntax. That is a great advantage. First, you get the power of SQL combined with the ease of use of object databases. Second, a new JPA developer with some experience with SQL can become productive very quickly.

The EntityManager object serves as a factory of 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 see another examples 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 single Double object reflecting the average x value of all the Point objects in the database.

Retrieving Existing Entities

The Retrieving Entities section of this manual describes several methods to retrieve 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. The TypedQuery interface, which was introduced in JPA 2, is a type safe subinterface of Query, which should usually be preferred. 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 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 details.

Updating and Deleting Entities

JPA refers to entity objects that are associated with an EntityManager as managed. A new 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 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 to automatically detect changes and apply 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();
```

All the Point objects whose x coordinate is greater or equal to 100 are deleted. All the other Point objects are being updated.

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

1.4 What is Next?

This chapter introduces 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 try using ObjectDB first, you can now write and run your own first ObjectDB/JPA programs.

First Steps in using ObjectDB

For step by step instructions on how to use ObjectDB in your IDE you can follow a tutorial:

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

These tutorial show how to run the sample program of this chapter. You can easily start your own first ObjectDB/JPA programs 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 and JPQL

The last two chapters complete the picture by describing specific ObjectDB issues:

- Chapter 5 - Object Database Tools
- 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:

- JPA Persistable Types
- JPA Entity Fields
- JPA Primary Key
- Auto Generated Values
- Index Definition
- Database Schema Evolution
- JPA Persistence Unit

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 only be embedded in entity classes as fields.

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 class and methods can be final.
- Usually 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 as an 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 does not enforce these restrictions so mapped superclasses are treated by ObjectDB as ordinary entity classes.

Mapped superclasses are useful only in applications that use also an ORM 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 manage identity (primary key) of their own and that 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 details 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:

- Collections 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 of values of persistable types.

In addition to 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 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 supported collection type when the entity is retrieved. Actually, JPA requires declaring persistent collection and map fields only with 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 class.

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 exiting databases. 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 (0, 1, 2) and values of the `color3` field are stored internally as strings ("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 of 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 use only explicitly specified persistable types.

2.2 JPA Entity Fields

Fields of persistable user defined classes can be classified into the following five groups:

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

The first three groups (transient, persistent and inverse fields) are in use in both and . The last two groups (primary key and version fields) are in use only 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 (similarly 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 affects also serialization) or the JPA `@Transient` annotation (that 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 and 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. Please notice that persistent fields can be declared with any 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 and 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. Cascade settings are explained in chapter 3. Fetch settings are explained in the Retrieving Entities section. `null` values are allowed by default. Specifying `optional=false` (as demonstrated for `field1`) causes throwing an exception on any attempt to store an entity with `null` value in that field.

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 contains data that is not stored as part of the entity in the database, but is still available after retrieval by a special automatic query.

The following entity classes demonstrates a bidirectional relationships:

```
@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 `Department` entities. Instead, the set is automatically filled for a `Department` entity that has been retrieved from the database by 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 affects the database, since the other side is not stored and calculated by a query.

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 that case, two unidirectional unrelated relationships are managed between the `Employee` and the `Department` classes and the application is responsible to keep them synchronized.

Inverse fields may improve efficiency in managing very large collections that are changed often, 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 is 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 mapping of employee names to `Employee` objects.

Single value inverse field is 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 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 (which 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. Versions serve for optimistic locking (as explained in Lock Management section).

You can expose entity object versions, making their values accessible to your application, by defining version fields in your entity classes. A version field should have a numeric type and should be annotated with the `Version` annotation:

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

If a version field exists, ObjectDB automatically injects the version values of entity objects into that field. Version fields should be considered by the application as read only. 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 if a version field is defined or not. Therefore, optimistic locking is supported by ObjectDB also when a version field is not defined. Nevertheless, defining a version field has some advantages:

- The application becomes more portable (to ORM 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 of non enhanced entity classes unless a version field is defined. Version verification during merge is performed only if either the entity class is enhanced or a version field is defined, because otherwise the version value of the detached object that is being merged is unavailable.

Property Access

ObjectDB has to access fields in various occasions. For instance, when an entity is stored, data is extracted from the persistent fields and stored in the database and 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 the get methods and specifying the Access annotation:

```
@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 providers (e.g. 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 property access, the default field access mode should usually be 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.

Entity Identification

Primary key values are unique per entity class in the database but multiple entity objects of different entity classes can have the same primary key value. Every entity object can be uniquely identified and retrieved from the database by the combination of its type and its primary key, as shown in the Retrieving Entities section. Only entity objects have primary keys. Instances of other persistable types, including embeddable types, are always stored as part of containing entity objects and do not have identity of their own.

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 by ObjectDB into that field. The field type can be any Java primitive or wrapper numeric type (e.g. `long`, `Long`, `int`, ...).

The `@GeneratedValue` annotation specifies that the primary key is automatically allocated by ObjectDB. Automatic value generation is discussed in details 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
      :
}
```

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 have one of the supported types in the list 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 alternative 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;
}
```

In this form 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, the embeddable class functions also as an ID class (because it can represent complete primary key values), so a separate ID class is not needed.

Using Primary Key for 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 for 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 queries 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 and select the best solution.

2.4 Auto Generated Values

Marking a field with the `@GeneratedValue` annotation specifies that an automatic value has to be generated for that field. This is useful mainly for primary key fields but supported by ObjectDB also for ordinary persistent fields. Automatic value generation is available only for numeric fields. ObjectDB supports different strategies for automatic value generation as explained below.

The Auto Strategy

ObjectDB maintains a special global sequence for every database. That sequence is used to generate default primary key values for entities with no primary key fields, and for primary key fields that are annotated by `@GeneratedValue` with the `AUTO` strategy:

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

If no strategy is specified, `AUTO` is the default, so the following definition is equivalent:

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

The `AUTO` strategy generates an automatic value using the global sequence during commit for every new entity object. The generated values are unique at the database level.

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 sequence is managed per type hierarchy, so generated values are unique only per type hierarchy.

The Sequence Strategy

The `@SequenceGenerator` annotation (which can be specified on classes and fields) defines a sequence by

specifying a name, an initial value (the default is 1) and an allocation size (the default is 50). A sequence can be used (by one or more fields in one or more classes) by specifying the `SEQUENCE` strategy and the sequence name in the `@GeneratedValue` annotation:

```
@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). That might be useful when the primary key value is needed earlier. To minimize round trips to the database server every time a bulk of IDs is allocated (the number of IDs is specified by the `allocationSize` attribute). Not always all the allocated IDs are eventually used, so this strategy does not guarantees continues sequence values with no gaps.

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 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. To get 1 as the first generated value, `initialValue=1` has to be specified in `@SequenceGenerator` but `initialValue=0` has to be specified in `@TableGenerator`, because in TABLE strategy the table holds the last allocated ID.

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 the 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 (similarly to primary key). `@Index` represents either an ordinary index with no such constraint or also a unique index if `unique="true"` is specified (the default is false). Both can have an optional `name` attribute that has no specific role but might be presented in the 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 for primary key / version fields).

Composite Index

A composite index is an index on more than one persistent field. The `@Index` and `@Unique` annotations (with or without a `name` attribute) are also used for composite indexes, but in that case the names of the fields must be specified using the `members` attribute:

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

When the indexed fields are specified explicitly by using the `members` attribute, as shown above, the `@Index` or `@Unique` annotation can be attached to either the class or to any persistent field. In that case the position of the annotation has no effect.

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 equivalent to omitting `members` and attaching the `@Index` annotation 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, because 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

Every index is managed by ObjectDB has a BTree, which is an ordered map data structure that is adjusted to files (rather than memory based map data structures). The keys of the BTree are all the 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 consume maintenance time and storage space. Therefore, using indexes wisely require understanding of how ObjectDB use 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 these queries. In that case the order of the fields in the index is important. If x is the first, a range scan is supported. If y is the first, the BTree is not ordered by x values, so a full scan is required. 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 contains 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 lexicographic order of the words and is not ordered by the length of the words.

The bottom line is that every index that covers all the fields in the query enables query execution at least using a full index scan. Range index scan is more efficient but supported only if the index order matches.

ObjectDB uses indexes also for sorting results and for projection:

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

A composite index on fields x an y covers all the fields in the query (including the SELECT and ORDER BY clauses) saving the need to access the entities themselves. In this is case a full scan is required. Field y should be the first and field x should be the second in the composite index. This way, the results are produced in the requested order (the order of the scan), so separate sort of the results is not required.

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 of entity classes that do not change the persistent state of the entity class are transparent for ObjectDB. This includes adding, removing and modifying constructors, methods and non persistent fields. Modifications of the persistent fields of an entity class, however, do affect 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.

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 updated to the new schema only when that entity object is stored in 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 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 of 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 that must have a `name` attribute (“`my-pu`” in the example). The name identifies the persistence unit when it is used for instantiation of 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 in which managed classes can be searched.
- The `class` elements specify names of managed 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 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 JPA and JDO specifications and books.

Managed 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) registers automatically also 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:

- Main JPA Interfaces
- Working with JPA Entity Objects

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

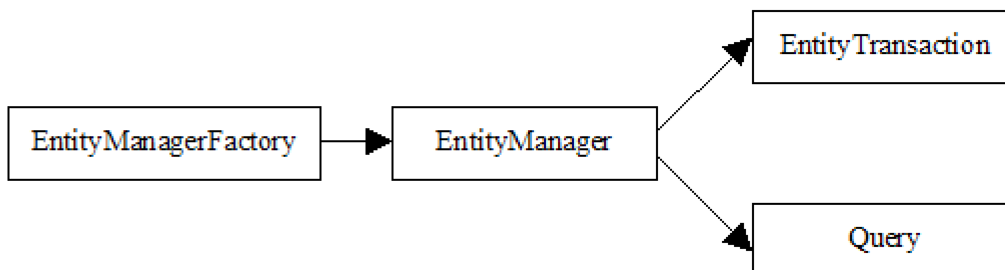
- CRUD Operations with JPA

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

- Advanced JPA Topics

3.1 Main JPA Interfaces

Working with the Java Persistence API (JPA) is based on 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.

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

An `EntityManager` instance also functions as a factory of `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, where the standard JPA interfaces keep your application portable.

EntityManagerFactory

An `EntityManagerFactory` instance is obtained by using a static factory method:

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

`Persistence` is a JPA bootstrap class that serves as a factory of `EntityManagerFactory`. 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 ends with `.odb` or `.objectdb` is considered by ObjectDB to be a database url rather than as a persistence unit name.

To use embedded mode, an absolute path or a relative path of a local database file has to be specified. To use client server mode, a url in the format objectdb://host:port/path has to be specified. In that case, an ObjectDB Database Server is expected to be running on a machine named host (could be domain name or IP address), listening to 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 path.

Another form of the `createEntityManagerFactory` method takes a map of properties as a second parameter. This form is useful when a user name and a password are required (in client-server mode) and no persistence unit is defined:

```
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 file and creates a . If the

database does not exist yet a new database file is created.

At the end of the work the EntityManagerFactory has to be closed:

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

Closing the EntityManagerFactory is required in order to close the database file and delete the recovery file (that should exist only while the database is open).

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 owner EntityManagerFactory instance. Calling the close method is essential to release resources (such as a socket in client-server mode) back to the owner EntityManagerFactory.

EntityManagerFactory defines another method for instantiation of EntityManager that 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.username", "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` class 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 either a call to `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 `rollback`, all the modifications to the database are discarded. Ending the transaction with `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.

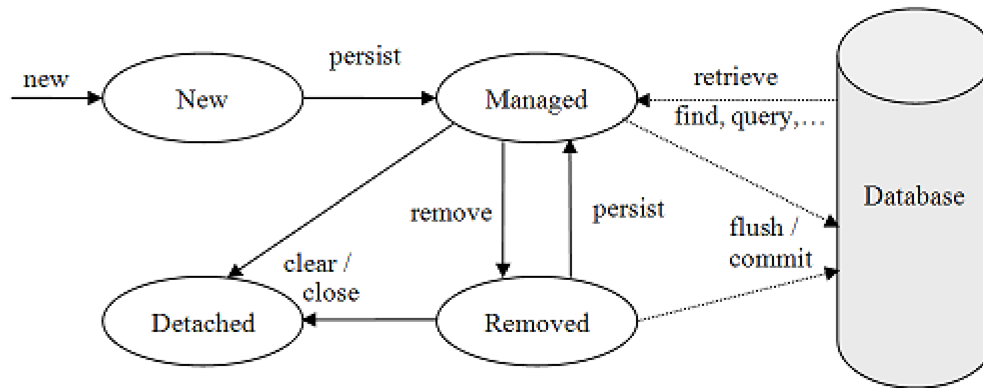
3.2 Working with JPA Entity Objects

Entity objects are instances of (persistable user defined classes), which can represent 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 you construct a new entity object its initial state is **New**. In that state the object is not associated with any `EntityManager` and represents nothing in the database.

You can store a New entity object in the database, by using the `EntityManager’s persist` method, which changes the state of a specified new object to **Managed** and associates it with a specific owner `EntityManager`. The `persist` method must be invoked within an active transaction. On transaction commit, the owner `EntityManager` stores the new entity object in the database. More details on storing objects are provided in the Storing Entities section.

`EntityManager` supports retrieval of entity objects from the database, for instance, by executing queries. Retrieved entity objects are constructed by ObjectDB with content form the database and their initial state is also **Managed**. Retrieving objects from the database is discussed in the Retrieving Entities section in more details.

If you modify a managed entity object within an active transaction, the change is detected by the owner `EntityManager`, and the update is propagated to the database on transaction commit. The Updating Entities section explains this issue.

You can also retrieve an object from the database and then mark it 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` that always requires accessing the database).

The main role of the persistence context is to make sure that within the same `EntityManager`, a database entity object is represented by no more than one memory entity object. 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 return the same memory entity object.

The persistence context functions also as a local cache at the `EntityManager` level. ObjectDB operates also 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 .

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:

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

All the managed entities in the persistence context become detached. Changes to entity objects that have not been flushed to the database are discarded. Detached entity objects are discussed in more details in the Detached Entities section.

3.3 CRUD Operations with JPA

This sections explains how to use JPA for CRUD operations:

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

3.3.1 Storing JPA Entitiy 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 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 by their own. Values of other persistable types can only be stored in the database embedded in containing entity objects (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 by either 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();
```

If Address is defined as an the Employee entity object is automatically stored in the database with its Address instance as an embedded object. Values of persistable types that are not entity classes (and instances of user defined embeddable classes are not different) are always stored automatically embedded in containing entity objects.

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 an entity class. In that case, if it was possible to store an Employee instance in the database without the referenced Address instance, the result would be a dangling reference in the database.

To avoid this undesired situation of a dangling reference, an IllegalStateException is thrown on commit if a persisted entity that should be stored in the database by the transaction references another entity object that is not expected to be stored in the database at the end of that specific transaction.

Entity objects whose state on commit is either New (i.e. the object has never been persisted) or Removed (i.e. the object is going to be deleted by this transaction) are not expected to be stored in the database, unless cascading persist is set.

Cascading Persist

Marking a reference field with CascadeType.PERSIST (or CascadeType.ALL that 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;
      :
}
```

The Employee entity class (above) contains an address field that references an instance of Address, which is another entity class. Persisting an Employee instance is automatically cascaded (because of the CascadeType.PERSIST setting) to the referenced Address instance, which is also persisted. Cascading

continues recursively when applicable (e.g. to entity objects that the Address object references, if any).

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 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();
```

Holding 1,000,000 managed Point instances in the persistence context might consume too much memory (managing entities consumes more memory than just the memory that is used directly by the Point instances). The sample code above clears the persistence context once in a while. The updates are flushed

to the database before clearing the persistence context, otherwise the updates would have been lost. The combination of `clear` and `flush` enables moving the updates from memory to the database.

Updates that are sent to the database using `flush` are considered as temporary and are visible (until commit) only to the caller `EntityManager`. With no explicit `commit`, these updates are later discarded. Flushing updates to the database is also useful 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.getTransaction().begin();
    }
}
em.getTransaction().commit();
```

In many cases using this solution (of multiple transactions) is preferred.

3.3.2 Retrieving JPA Entity Objects

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

The serves as a cache for retrieving objects. If the entity object that has to be retrieved is not found in the persistence context, a new object is constructed and filled with content that is retrieved from the database. Notice that construction of a new managed object during retrieval uses the no-arg constructor. Therefore, it is recommended to avoid time consuming operations in no-arg constructors of entity classes, and to keep them simple. The new entity object is added to the persistence context as a managed entity object and then returned to the application.

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 to Employee is not required because find is defined as returning an instance of the 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 managed object is returned. Otherwise, the object content is retrieved from the database. If the object is not found in the database null is returned. If the object is found a new managed object with the retrieved content is constructed and returned.

A similar method, getReference, can be considered as 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 object does not exist an EntityNotFoundException is thrown when the retrieval fails.

The getReference method is useful when a reference to an entity object is required but not its content, for example, when setting a reference to it 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 usually is unacceptable.

A persistent reference field can be excluded from this automatic cascaded retrieval:

```
@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 cascading the retrieval operation.

When an entity object is retrieved all its persistent fields are initialized. A persistent reference field with 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.

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, other entity objects that it references through collections and maps are not retrieved immediately 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) includes the field in cascading the retrieval operation. In the above example, when an Employee instance is retrieved, all the referenced Project instances are also retrieved with it.

Retrieval by Navigation and Access

When an entity object is retrieved from the database and the EntityManager is still open, all the persistent fields can be accessed freely, regardless of the current fetch policy. This also includes fields that reference other entity objects that have not been loaded from the database yet and are represented by managed hollow objects. If the EntityManager is open (regardless if there is an active transaction or not) 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 regardless the fetch policy. This illusion, which is based on lazy transparent activation of objects by ObjectDB, helps hiding some of the direct interaction with the database and makes database programming easier.

For example, after retrieving an `Employee` instance from the database, its `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 the name of the `manager` causes retrieval its content from the database and initialization of all its persistent fields.

As seen, the entire graph of objects is available for navigation, regardless of the fetch policy. The fetch policy, however, do affects performance. Eager retrieval might minimize the round trips to the database and improve performance. On the other hand, unnecessary retrieval of entity objects that are not in use decreases performance.

In addition, if entity objects become Detached (e.g. when the `EntityManager` is closed) transparent activation, which requires an open `EntityManager`, is not supported and then only content that has already been fetched from the database is available.

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 queries. This issue is explained in details in the JPA Queries and JPQL chapter.

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 fresh content that is retrieved from the database. This might be useful to assure that the application deals with the most up to date version of an entity object, in case it might have been changed by another `EntityManager` since it has been retrieved.

An `IllegalArgumentException` is thrown by `refresh` if the argument is not a managed entity (including entity object in New, Removed or Detached modes). 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` that covers `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.<code>REFRESH</code>)
    private Address address;
     :
}
```

The `Employee` entity class (above) contains an `address` field that references an instance of `Address`, which is another entity class. Refreshing an `Employee` instance is automatically cascaded (because of the `CascadeType.REFRESH` setting) to the referenced `Address` instance, which is also removed. Cascading continues 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

After retrieving an entity object from the database (no matter which way), it can simply be modified in

memory while a transaction is active:

```
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 is cascaded from all the entity objects that have to be stored in the database, including 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 is enabled, all the reachable entity objects that are not managed yet are persisted.

Automatic Change Tracking

As shown above, update is achieved by modifying a managed entity object 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 much more efficient way is based on enhancing entity classes.

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, unless the array field itself is set:

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

em.getTransaction().begin();
employee.projects[0] = new Project(); // not detected
employee.projects = employee.projects; // detected
em.getTransaction().commit();
```

As demonstrated above, setting an array cell is not detected automatically but assignment of the array field to itself is detected. Instead of using this workaround of touching the array field, ObjectDB can be to detect array cell changes using snapshots, also when enhanced entity classes are in use.

It is usually recommended to prefer collections over 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, first it must be retrieved (no matter which way), and then, when a transaction is active, 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 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` that covers `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.<code><a
href="/api/java/jpa/CascadeType/REMOVE"
_mce_href="/api/java/jpa/CascadeType/REMOVE">REMOVE</a></code>)
```

```
    private Address address;
     :
}
```

The `Employee` entity class (above) contains an `address` field that references an instance of `Address`, which is another entity class. Removing an `Employee` instance is automatically cascaded (because of the `CascadeType.REMOVE` setting) to the referenced `Address` instance, which is also removed. Cascading continues 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 `orphanRemovel` 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 aspect, `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 the response to disconnecting a relationship. For example, 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 a 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 issues:

- Detached Entity Objects
- Locking in JPA
- JPA Lifecycle Events

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 database objects. Compared to managed entity objects, detached objects are very limited in functionality:

- Many JPA methods do not accept detached objects (e.g. `lock`).
- Only persistent fields of detached objects that have been loaded when they were managed contain valid data. is not supported.
- Changes to detached entity objects are not stored in the database (unless the detached entity objects are merged and become managed again).

Detached entity objects are useful when an `EntityManager` is not available and in 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 during deserialization as a detached entity object, since it is not

associated with any EntityManager.

In addition, starting JPA 2, an entity object can also be detached by 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 that covers 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;
     :
}
```

The Employee entity class (above) contains an address field that references an instance of Address, which is another entity class. Detaching an Employee instance is automatically cascaded (because of the CascadeType.DETACH setting) to the referenced Address instance, which also becomes detached. Cascading continues 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 the managed entity objects from that EntityManager:

- 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 that covers 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;
     :
}
```

The Employee entity class (above) contains an address field that references an instance of Address, which is another entity class. Merging an Employee instance is automatically cascaded (because of the CascadeType.MERGE setting) to the referenced Address instance, which is also merged. Cascading continues recursively when applicable (e.g. to entity objects that the Address object references, if any).

3.4.2 Locking in JPA

ObjectDB supports both JPA **optimistic locking** and JPA 2 **pessimistic locking**. Locking is essential to avoid update collision as a result of simultaneous update of 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 acts 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 based on an old version of a database

object, for which an update has already been committed by another user.

When using ObjectDB, optimistic locking is always active 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. Using pessimistic locking, database objects are locked during a transaction, lock conflicts can be detected earlier, and less surprises are expected at transaction commit time.

Optimistic Locking

ObjectDB maintains a version number for every entity object. The initial version of a new entity object (which 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 .

During `commit` (and `flush`), ObjectDB checks every database object that has to be updated or deleted, by comparing the version number of that object in the database to the version number on which the update is based. The transaction fails and an `OptimisticLockException` is thrown if the version number mismatches, indicating that the object has been modified by another user (using another `EntityManager`) since it has been retrieved by the current updater.

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

Pessimistic Locking

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.

The main supported pessimistic lock modes are:

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

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) already holds a PESSIMISTIC_WRITE lock on that database object.
- A PESSIMISTIC_WRITE lock request fails if another user already holds either a PESSIMISTIC_WRITE lock or a PESSIMISTIC_READ 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 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, should cause a LockTimeoutException (by the second lock request).

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

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, it is silently ignored by ObjectDB.

The OPTIMISTIC_FORCE_INCREMENT mode affects only a clean (non dirty) entity object, which is marked as dirty and its version number is increased by 1, as a result of lock in that mode.

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 on Retrieval

JPA 2 provides various methods for locking entity object when they are retrieved from the database. In addition to improving efficiency, these methods enables combining retrieval and lock in one atomic operation.

For example, the `find` method has a form that receives also 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 in order to lock all the query result objects.

3.4.3 JPA Lifecycle Events

Callback methods are user defined methods that are invoked automatically by ObjectDB (following JPA rules) as a result of entity lifecycle events.

Internal Callback Methods

Internal callback methods are defined within the entity class:

```
@Entity
public static class EntityWithCallbacks {
    @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` - when 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. 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 invoked also for entity objects of the subclasses, except if the callback method is overridden in a 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 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 Listener {
    @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 instantiation by ObjectDB.

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

```
@Entity @EntityListeners(Listener.class)
public class EntityWithListener {
}
```

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

```
@Entity @EntityListeners({Listener1.class, Listener2.class})
public class EntityWithTwoListeners {
}
```

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 apply by default to all the entity classes. Default listeners must be specified in a mapping XML file, because currently there are no equivalent annotations:

```
<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 be invoked for a single 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 any internal callback method (which is defined in an entity class).
- 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.

Chapter 4 - JPA Queries and JPQL

ObjectDB supports two query languages:

- JPQL - Java Persistence (JPA) Query Language
- JDOQL – Java Data Objects (JDO) Query Language

JPQL can be considered as an object oriented version of SQL. Users that are familiar with SQL should find JPQL very easy to use. On the other hand, JDOQL is more Java oriented and is based on the syntax of Java. Both query languages support clauses such as ORDER BY, GROUP BY and HAVING. Mixed JPQL/JDOQL queries are also supported by ObjectDB, regardless of which API (JPA or JDO) is used.

This chapter focuses on using queries in JPA using 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:

- The JPA Query API

The Java Persistence Query Language (JPQL) is discussed in two other sections:

- JPQL Query Structure
- JPQL Expressions

The first section explains the main clauses of JPQL queries (SELECT, FROM, etc.), and the second section explains how to use JPQL expressions to build these clauses.

4.1 The JPA Query API

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

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, which is usually the case, 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 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`:

```
  <a href="/api/java/jpa/Query">Query</a> q1 = em.createQuery("SELECT c FROM
Country c");

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

The same JPQL query, which retrieves all the `Country` objects in the database, is represented by both `q1` and `q2`. When building a 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.

JPA 2 introduces also a third way for building queries, using the Criteria API, which is expected to be supported by future ObjectDB versions.

Further Reading

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

- Running JPA Queries
- Query Parameters in JPA
- Setting & Tuning of JPA Queries
- JPA Named Queries

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

- JPQL Query Structure
- JPQL Expressions

4.1.1 Running JPA Queries

The `Query` and `TypedQuery` interfaces provide two methods for running SELECT queries:

- `Query.getSingleResult` and `TypedQuery.getSingleResult` -
  for use when **exactly one** result object is expected.

- `Query.getResultList` and TypedQuery.getResultList - for general use in any other case.

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 by 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` provide a `getResultList` method, but the version of `Query` returns a raw type (non generic) result list instead of 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>`) causes a compilation warning (which can be suppressed by specifying `@SuppressWarnings("unchecked")`). If the new `TypedQuery` interface is used whenever a known result type is expected, casting and warning are avoided.

The query result collection functions as any other ordinary Java collection. A parameterized type result enables easy iteration by the enhanced (foreach) loop:

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

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

Single Result Query Execution (with getSingleResult)

The `getResultList` method (which is discussed above) can also run queries that return a single result object. In that 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 the `getSingleResult` method, as a more convenient method for the cases in which exactly one result object is expected.

By definition, 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();
```

The above 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 = 'Canada'");
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.

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 repeating 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 (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.

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 embed the name as a `String` literal. There are a few drawbacks in 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. Notice, that 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 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.

Third, parameters are more flexible and supports elements that are unavailable as literals, such as entity objects.

API Parameter Methods

In JPA 2 the `Query` interface defines 18 methods for handling parameters, where 9 of them are also overridden in `TypedQuery` (to support method chaining). That is more than a half of the total number of methods in `Query` and `TypedQuery`. That large number of methods is not typical to JPA, which generally excellences in its thin and simple API.

There are 9 methods for setting parameters in a query (which is essential whenever using query parameters) and there are 9 methods for extracting parameter values from a query (which is new in JPA 2 and expected to be less commonly used than the setting 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 9 setting methods rather than 3 is that JPA provides additional 3 special methods for setting `Date`

parameters and additional 3 special methods for setting Calendar parameters.

Date and Calendar parameter values require special methods, in order to specify what they represent, a pure date, a pure time or a combination or date and time, as explained in details in the section.

For example, the following invocation passes a Date object as a pure date:

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

Since TemporalType.Date represents a pure date, the time part of the new constructed java.util.Date instance is discarded. This is very useful in comparison against a specific date, where 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), with or without 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 as a value (isBound). These methods are not required for running queries and are expected to be less commonly used.

4.1.3 Setting & Tuning of JPA Queries

The Query and TypedQuery interfaces define various setting and tuning methods that affect query execution. As with setting parameters, these setting methods can be invoked anytime before executing the query with getResultList or getSingleResult.

Result Range (setFirstResult, setMaxResults)

The setFirstResult and setMaxResults methods support defining a result window that exposes a section 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 in the complete original 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 supports implementing result paging efficiently. 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 results of a specified page:

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

```
            .setMaxResults(pageSize)
            .getResultList();
```

These methods can be invoked in a single expression with `getResultList` because all the setting 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`, are visible to anyone who uses `em` also before committing the transaction (but not to users of other `EntityManager` instances). For simple JPA operations (e.g. `find`) this is handled automatically by JPA implementations and by ObjectDB by considering local transaction changes. Query execution is much more complex. Therefore, before a query is executed, uncommitted database changes (if any) are automatically flushed to the database.

Flushing to database affects performance. Therefore, it is more efficient to execute queries on a clean `EntityManager` that does not hold uncommitted database changes.

Alternatively, application can ask JPA to avoid flushing before query execution to improve efficiency. This is possible whenever it is known that the changes do not affect the query results or when the effect can be ignored.

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

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

The default is `AUTO`, but that 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, overriding the default `EntityManager` setting:

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

// 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, even though keeping the default `AUTO` flush mode is easy and safe, when performance is important, this issue has to be considered.

Lock Mode (setLockMode)

ObjectDB uses automatic to prevent concurrent changes to entity objects by multiple users. JPA 2 adds support for . 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 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`.

4.1.4 JPA Named Queries

A named query is a statically defined query, with a predefined unchangeable query string, which is accessed at runtime by its unique name. Using named queries instead of dynamic queries may improve code organization by separating the JPQL query strings from the Java code. It also enforces using query parameters rather than embedding literals dynamically in the query string, which 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 elements are required and two elements are optional. The two required elements, name and `query`, are shown above and define the name of the query and the query string itself. The two optional elements, `lockMode` and `hints`, provide static replacement for the `setLockMode` and `setHint` methods.

Every `@NamedQuery` annotation has to be applied to exactly one entity class or mapped superclass. Usually, every named query is attached to the most relevant entity class. However, 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 (as a replacement to 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 when using ObjectDB, this issue is not covered here.

Using Named Queries at Runtime

Named queries are represented at runtime by the same `Query` and `TypedQuery` interfaces. The only difference is the `EntityManager` factory methods, which receive a query name rather than a query string as a parameter. The `createNamedQuery` method that receives also a result type returns a `TypedQuery` instance:

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

Where the `createNamedQuery` method with one parameter returns a `Query` instance:

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

One of the reasons that JPA requires 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 classes must be available.

ObjectDB makes the definition of a persistence unit optional. Named queries are automatically searched in all the managed classes that ObjectDB knows, 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 setting the special property:

```
em.setProperty("objectdb.user.classes", new Class[] { MyEntity.class });
```

ObjectDB will include all the classes that are specified in that property in the named queries search.

4.2 JPQL Query Structure

The syntax of the Java Persistence Query Language (JPQL) is very similar to the syntax of SQL. Having an SQL like syntax in JPA queries is an important advantage because SQL is very powerful and because many developers are already familiar with SQL.

The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, where 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 Format

As with SQL, a JPQL 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 mandatory in every retrieval query (update and delete queries have a slightly different form). The other clauses, WHERE, GROUP BY, HAVING and ORDER BY are optional.

In general, 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. Naturally, 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 referenced 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 above query, 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 is actually all the Country objects in the database.

Organization of this Section

This section contains the following pages:

- JPA/JPQL SELECT
- JPA/JPQL FROM
- JPA/JPQL WHERE
- JPA/JPQL GROUP BY and HAVING
- JPA/JPQL ORDER BY

4.2.1 JPA/JPQL SELECT

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 to other database objects, , 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 results. Specifying the required query results more precisely can improve performance and in some cases can also reduce Java code. 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 referenced to as projection. The field value is 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 values 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 be returned also from a query that uses projection, when a result path expression is resolved 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 modifying them when a transaction is active does not commit changes 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 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 is the same multi SELECT expression query that was shown above, but now the query results are `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;
    }
}
```

As expected, this query returns a list of CountryAndCaiptal instances:

```
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 only serves for collecting and processing query results.

If an entity class is used as a result class, the result entity objects are created in the , 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 'I%'
```

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 'I%'
```

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.

4.2.2 JPA/JPQL FROM

The FROM clause declares query identification variables that define iteration over objects in the database. A query identification variable is similar to a variable of a Java enhanced for (foreach) 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,

using a .

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 in the @Entity's name annotation element.

Multiple range variables are also 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 iterate also 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 avoid 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 variables, join variables, 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. The new query form that uses a join variable is preferred. Besides being shorter and cleaner, the new form 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 that 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, which 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. Therefore, it is completely excluded from 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. Therefore, a `Country` c with no `Capital` city has a minimum representation in the FROM iteration, in a (c, NULL) pair.

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.

Reserved Identifiers

A name of a JPQL query variable must be valid Java identifiers, which is not one of the following reserved identifiers:

```
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.

4.2.3 JPA/JPQL WHERE

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 a 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 is evaluated to TRUE are passed by 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 , filtering is done earlier during FROM iteration. In our example, if an index is defined on the `population` field - ObjectDB can use that index to iterate directly only on `Country` objects that satisfies 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 or not.

WHERE Filter in Multi Variable Queries

In a multi variable query, the FROM clause defines iteration on tuples. In that 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, which 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 by 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,
that 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 the other JPQL query clauses but they are especially dominant in the WHERE clause.

4.2.4 JPA/JPQL GROUP BY and HAVING

The GROUP BY clause enables grouping of query results. A JPQL query that has a GROUP BY clause returns properties of generated groups instead of individual objects and properties.

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 to 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 passing the groups to the SELECT clause, which return the first letters as result.

Please notice that ObjectDB is very flexible in using JPQL expressions anywhere in the query. This query

might not work with some JPA providers. Currently only identification variables and path expressions are supported in the GROUP BY clause by all the JPA implementations.

Grouping the `Country` objects make 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, mac, 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 as merely a DISTINCT operator. The above GROUP BY query (which does not use aggregates) can be replaced by 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 the number of elements as a `long` value.
- `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 the number of countries with names that starts with each 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 as results `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 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 is useful for COUNT aggregate function, for eliminating duplicate values when counting.

Only the COUNT aggregate function can be applied to entity objects directly. Other aggregate functions are applied to properties of objects in the group, by using a path expressions.

The following query groups countries in Europe by their currency, and for each group retrieves 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 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 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 apply also to the HAVING clause, which means that individual object properties 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 the 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 calculated. Any JPQL query that contains an aggregate expression in the SELECT clause is considered as a grouping query, with all the involved restrictions, even when a GROUP BY clause is not specified. Therefore, only aggregate functions can be specified in the SELECT clause in that case - individual objects become inaccessible and group properties, i.e. aggregate expressions are supported.

4.2.5 JPA/JPQL ORDER BY

The ORDER BY clause enables ordering of query results. The order of the results in any JPQL query that does have an ORDER BY clause is undefined and might depend on the specific query execution program that is used for executing the query.

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 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 to collect as results. Then the SELECT clause is used to build the results by evaluating the result expressions. Finally the results are ordered by evaluation of the the order 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 not valid, 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
```

ObjectDB support a very flexible ORDER BY clause in which every JPQL expression can be used, as long as its type is comparable (i.e. numbers, strings and date values). Some JPA implementation are more restrictive. Path expressions as used in the code above are always supported but support of other JPQL expressions is vendor dependent.

Query results can also be ordered by multiple ordering expressions. In that case, the first order expression serves as the primary ordering expression. Any additional ordering expression is used to order results for which all the previous ordering expressions produce the same value.

The following query returns `Country` objects ordered, first by `currency` and then by `name`:

```
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 referenced to as result variables, to distinguish them from the identification variables that are defined in the FROM clause.

ORDER BY 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 get the query results order in descending order - the DESC keyword must be added explicitly to the relevant ordering expression:

```
SELECT c.name FROM Country c ORDER BY c.name DESC
```

ORDER BY in Grouping Query

The ORDER BY clause is always the last in the query processing chain. When both ORDER BY and GROUP BY clauses are defined - 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
```

Without an ORDER BY the result order is undefined. the ORDER BY clauses orders the returned results by the currency name.

4.3 JPQL Expressions

JPQL expressions are the foundations on which JPQL queries are built.

A JPQL query consists of clauses - SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY, where each clause consists of JPQL expressions.

Atomic Expressions

The JPQL atomic expressions are:

- JPQL Variables
- JPQL Parameters
- JPQL Literals

Every JPQL expression consists of at least one atomic component. More complex JPQL expressions are built

by combining atomic expressions with JPQL operators and functions.

Operators and Functions

JPQL supports the following operators (in order of decreasing precedence):

- :
  Unary operators: + (plus) and - (minus).
  Binary operators: * (multiplication) and / (division).
  Binary operators: + (addition), - (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, JPQL supports predefined functions, which are also described in this section.

Organization of this Section

This section contains the following pages:

- JPQL Literals
- Object Expressions in JPQL
- Numbers in JPQL
- Strings in JPQL
- Collections in JPQL
- JPQL Comparison Operators
- JPQL Logical Operators

4.3.1 JPQL Literals

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 moderately. A query that uses parameters instead of literals is more generic and efficient, because it 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 in 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 supports also hexadecimal numeric literals (e.g. 0xFF, 0xFFL) and octal numeric literals (e.g. 077, 077L), but it is currently not supported by all the 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 supports also 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 presents data) are case sensitive, so 'abc' and 'ABC' are not equivalent.

Date 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 '2010-12-31 23:59:59'}

In addition, JPA defines special JPQL date expressions that represent the date and time on the database server when the query is executed (and per query execution act as literals):

- CURRENT_DATE - the current date (a `java.sql.Date` instance)
- CURRENT_TIME - the current time (a `java.sql.Time` instance)
- CURRENT_TIMESTAMP - current time stamp (a `java.sql.Timestamp` instance)

When using ObjectDB in client-server mode (rather than embedded mode) these expressions are evaluated to the data and time on the server.

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 com.objectdb.ui;

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

Then `com.objectdb.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, similarly 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, as explained in Objects in JPQL.

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 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.

4.3.2 Object Expressions in JPQL

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 objects of JPA managed classes are assigned as arguments).
- Path expressions that navigate from one user 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 the 6 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 using 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 how exactly it works by considering the equivalent JOIN query as explained in the JPQL FROM page.

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 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 reference 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 a - all the objects are considered as one group, and COUNT calculates its size).

4.3.3 Numbers in JPQL

Numeric values may appear in JPQL queries in many forms:

- as (e.g. 123, `-12.5`).
- as parameters (where numeric values are assigned as arguments).
- as path expressions (where navigation is to a persistent numeric field).
- as aggregate expressions (e.g. COUNT).
- as collection functions (e.g. INDEX, SIZE).
- as string functions (e.g. LOCATE, LENGTH).
- as composite arithmetic expressions that use arithmetic 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 supports also the `%` (modulo) and the `~` (bitwise complement) operators, following Java and JDO. JPA follows Java numeric promotion principles. For example a binary arithmetic operation on an `int` value and a `double` value results in a `double` value.

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.

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 division of one number by another, similarly to the modulo operator (%) in Java (which is also supported by ObjectDB as an extension).

For example:

- `MOD(11, 3)` is evaluated to `2`
- `MOD(8, 4)` is evaluated to `0`

The MOD functions takes two integer values of any types 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.

4.3.4 Strings in JPQL

String values may appear in JPQL queries in various forms:

- as (e.g. `'abc'`, '').
- as parameters (where string values are assigned as arguments).
- as path expressions (where navigation is to a persistent string field).
- 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 for check (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:

- `'100%' LIKE '%\%' { ESCAPE '\' }` is evaluated to TRUE.
- `'100' LIKE '%\%' { ESCAPE '\' }` is evaluated to FALSE.

Only the first percent character (%) in the two above expressions is a wildcard. The second one is escaped and represents matching against a real % character.

LENGTH - Counting Characters in a String

The LENGTH(str) function returns the number of characters in the string as an `int` value.

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`.

The third argument (when exists) specifies from which position to start the search. 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.

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 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 (in the third optional argument), the entire string suffix, starting at the specified position, is returned.

Java String Methods

ObjectDB supports also Java String methods (as part of its support in JDO), but using these methods in JPQL queries in not portable.

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).

4.3.5 Collections in JPQL

Collections may appear in JPQL queries:

- as parameters (where collections are assigned as arguments).
- as path expressions (where navigation is to a persistent collection field).

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.

[NOT] MEMBER OF

The `[NOT] MEMBER OF` operator checks whether if a specified element is contained in a specified collection.

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'`.

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.

4.3.6 JPQL Comparison Operators

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 -JDO / Java
Less Than	<	<
Greater Than	>	>
Less Than or Equal To	<=	<=

	Set 1 - JPQL / SQL	Set 2 -JDO / Java
Greater Than or Equal To	>=	>=
Equal	=	==
Not Equal	!=	<>

The difference between the two sets is only in the Equal and the Not Equal operators. JPQL follows the SQL notation, where Java uses its own notation (which are 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 represents one `NULL` value (either on the left or right side, where on the other side there is a non `NULL` value) and the other column represents 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 expression that includes an unknown value is 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 (=, <>, ==, !=), also, following the logic of `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 (=, <>, ==, !=). Note that comparison for these classes follow the logic of == in Java rather than of `equals`.

ObjectDB supports comparison of any two values that belong to the same group according to the above division. Therefore, 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 identical:

```
x BETWEEN :min AND :max
```

```
x >= :min AND x <= :max
```

Similarly, NOT BETWEEN is also a shortcut and the following expressions are identical:

```
x NOT BETWEEN :min AND :max
```

```
x < :min OR x > :max
```

4.3.7 JPQL Logical Operators

By using logical operators in JPQL queries - complex JPQL boolean expressions can be composed 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 - JDO / Java
AND	&&
OR	||
NOT	!

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 preceding of the JPQL NOT operator and the ! Java/JDO operator.

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.

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 of OR.

Unary NOT (!) Operator

The following query retrieves all the countries which population does not exceed a specified limit:

```
SELECT c FROM Country c
WHERE NOT (c.population > :population)
```

The operand of a NOT operator must always 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 supports also the Java/JDO `!` operator as a replacement of `NOT`.

Chapter 5 - Object Database Tools

This chapter explains how to use the following ObjectDB tools:

- ObjectDB Explorer
- ObjectDB Server
- ObjectDB Enhancer
- ObjectDB Doctor
- ObjectDB Replayer

5.1 ObjectDB Explorer

ObjectDB Explorer is a tool that enables viewing the content of ObjectDB Databases visually.

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 setting the paths to the `objectdb.jar` file and to the JVM).

Opening a Database

To open a local database file use the `[File > Open Local...]` menu command (or the equivalent toolbar button) and in the dialog box select the desired local database file

Recently used local database files can also be opened using the `[File > Recent Local Files]` menu command. By default, when the Explorer starts it opens the last used local database file automatically. You can change this behavior in `[Tools > Options > General]`.

To open a remote database using client-server mode use the `[File > Open Remote...]` menu command

(or the equivalent toolbar button). In the dialog box you have to fill in the host, port, username and password of the remote connection. You also have to specify a database path on the remote server, possibly using the "Browse" button.

Use the `[File > Close]` menu command to close a local database file or a connection to a remote database file.

Viewing Database Content

The Explorer provides two types of viewer windows for exploring database objects.

The **Table** window displays a collection of objects following 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 the data of a simple object model.

In most cases, however, the **Browser** window (which is designed to handle more complex object models) is preferred. The browser window displays objects using a tree. Every database object is represented by a tree node, and the values of its persistent fields are represents by child nodes. The main advantage of using the browser window is in navigation among objects. 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 might be accessed using different paths in the tree and may therefore 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 identified as a descendant of itself in the tree).

To open a new viewer window, first select the target element - an entity class in the "Class" tabbed window or an object in a currently open table or browser viewer. When the target element is selected, you can open a new viewer window using the `[Window > Open Table Window]` or the `[Window > Open Browser Window]` menu commands, or by the equivalent context menu commands (right clicking the selected target element) or the equivalent toolbar buttons. When the target element is an object in an open viewer window, the `[Window > Open Focus Selection]` command switches the current viewer to focus on the selected object. A double click on the target element opens the browser window by default (you can change the default to a table window in `[Tools > Options > General > Display]`).

5.2 ObjectDB 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 on other remote machines, using the client-server mode.

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:

```
> java com.objectdb.Server start
```

The Server configuration is loaded automatically as explained in the Configuration section. 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 be used also 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 located in the `bin` directory. For this to work, the original structure of ObjectDB directory must be preserved because `server.exe` tries to locate and load the objectdb.jar at the same directory.

By default, `server.exe` starts the server using the following command:

```
> java -server -Xms32m -Xmx512m com.objectdb.Server run
```

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 run
```

The `server.exe` application is represented, when running, 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 application.

5.3 ObjectDB Enhancer

ObjectDB Enhancer is a post compilation tool that improves performance by modifying byte code of compiled classes after compilation. Enhancement is mainly for user defined persistable classes (, and), and usually optional.

In once case enhancement is required. Non persistable classes that access directly (not through methods) persistent fields of enhanced classes, must be also enhanced. It is a good practice (and actually required by JPA but not enforced by ObjectDB) to avoid accessing persistent fields from other classes directly (e.g. by using `get` and `set` methods). If you Follow this practice, only user defined persistable classes have 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 Updating Entities section).
- 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.

Build Time 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 ObjectDB `bin` directory. You might need to edit the paths to the `objectdb.jar` file and to the JVM in the provided shell script.

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 are searched also in sub directories:

```
> 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 in 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 a name of a persistence unit to enhance all the managed classes that are included 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, keeping the original files unchanged:

```
> java com.objectdb.Enhancer -s "*.class" -d enhanced
```

Finally, build time enhancement can also be invoked from 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, but as one 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 enhancement 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, assuming the `objectdb.jar` file 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 should 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 , ObjectDB tries to load the Enhancer as a Java Agent and enhance classes on the fly during load, even if a Java Agent not specified explicitly.

This enhancement technique is inferior to the other techniques that are described above. First, currently it works only on 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 ObjectDB 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 if exists (or copying, moving or transferring an ObjectDB database file without its recovery file).
- 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), bypassing ObjectDB internal file lock protection.
- Modifying the database file externally not through ObjectDB (e.g. by a malicious software such as a computer virus).

Giving 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 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, by:

```
> 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 (non existing yet) database file that is generated by the ObjectDB Doctor.

5.5 ObjectDB Replayer

ObjectDB can record its internal engine operations in special binary recording (journal) files. Recording is disabled by default and should be enabled in the when needed.

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 , 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 .

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 exist. 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, by:

```
> 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 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 result 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 specified run above succeed and all the operations until transaction 1000 are applied,
the generated result file is expected to be 1000.odb.

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 and Logging - <general>
- Database Management - <database>
- Entity Management - <entities>
- Schema Update - <schema>
- Server Configuration - <server>
- Server User List - <users>
- SSL Configuration - <ssl>

This page explains general ObjectDB configuration issues.

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 alternative path by setting the `"objectdb.conf"` system property:

```
  System.setProperty("objectdb.conf", "/my/objectdb.conf");
```

As with any other system property, 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 in which ObjectDB is run).

6.1 General and Logging - <general>

The <general> configuration section specifies ObjectDB settings that are relevant to both the server side and the client side.

The default configuration file contains the following <general> section:

```
<general>
  <temp path="$temp/ObjectDB" threshold="64mb" />
  <network inactivity-timeout="0" />
  <log path="$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 <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 - <database>

The `<database>` configuration section 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>` section:

```
<database>
  <size initial="256kb" resize="256kb" page="2kb" />
  <recovery enabled="false" sync="false" path="." max="100mb" />
  <recording enabled="true" sync="false" path="." mode="write" />
  <processing cache="64mb" max-threads="10" synchronized="false" />
  <query-cache results="32mb" programs="500" />
</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="false" sync="false" path="." max="8mb" />
```

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 loading 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).

Starting ObjectDB 2.0 RC3 recovery is disabled by default in favour of ObjectDB Replayer tool. Recording provides an alternative (which is usually 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 on or off.
- The `sync` attribute (whose value is `"true"` or `"false"`) specifies if 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 <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 space that should be available for caching pages of the database file (ObjectDB might use more space when necessary).
- 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. This optimal number should usually be 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 still have to be executed, so they 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).

6.3 Entity Management - <entities>

The <entities> configuration section specifies front end settings, which are relevant on the client side (and in embedded mode).

The default configuration file contains the following <entities> section:

```
<entities>
  <enhancement agent="false" reflection="warning" />
  <cache ref="weak" level2="0mb" />
  <cascade-persist always="auto" on-persist="false" on-commit="true" />
  <dirty-tracking details="false" 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 if 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 penalty. 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 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 hold 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 L2 cache can be disabled by specifying 0 or 0mb.

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 if cascading (as a result of either global or local setting) should be applied during persist.
- The on-commit attribute specifies if 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 <dirty-tracking> element specifies special dirty tracking issues:

- The arrays attribute specifies if modifications to array cells should be tracked automatically also in enhanced classes. See the Updating Entities section for more details.

6.4 Schema Update - <schema>

The <schema> configuration section 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 refactory. Only these schema changes are specified in the configuration file, other schema changes are

handled 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 configuration that matches these changes exactly.

The default configuration file contains an empty `<schema>` section. If that configuration section 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>` section 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 may have two roles:

- If the optional new-name attribute is specified - the package name is changed from its original name, which is specified by the required name attribute, to the new name. All the classes in that package are also affected, since their fully qualified names are changed.
- 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 may have two roles:

- If the optional new-name attribute is specified - the class name is changed from its original name, which is specified by the required name attribute, to the new name.
  The value of the name attribute must be unqualified (no package name) - the package name as specified in the containing <package> element is used.
  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 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 - <server>

The <server> configuration section specifies settings for running an ObjectDB Server.

The server is affected also by other sections in the configuration file, particularly the <users> and the <ssl> configuration sections.

The default configuration file contains the following <server> section:

```
<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 (as explained in the JPA Overview section) when connecting to the database.

- 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 some 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 can be managed by the server. The $objectdb prefix can be used to represent the ObjectDB installation directory, as demonstrated above.

The data path of an ObjectDB server is similar to the document root directory of a web server. Every database file in the data root directory including in subdirectories can be accessed by the server. Appropriate file system permissions have to be set on the root data directory (and on subdirectories and files) to enable operations of the server process.

When connecting to the server, the path that is specified in the url connection is resolved relatively to the root data directory. For example, "objectdb://localhost/my/db.odb" refers to a database file db.odb in a subdirectory my of the root data directory.

6.6 Server User List - <users>

The <users> configuration section 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> section:

```
<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 super user. A super user is authorized to manage server settings using the ObjectDB Explorer.

A value `"$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 root 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 - <ssl>

The <ssl> configuration section 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> section:

```
<ssl enabled="false">
  <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>
```

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 should 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 attribute, `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).