# Contents

## 1 Overview

1.1 What is JPA? .......................................................... 1
1.2 Why use JPA? .......................................................... 1
1.3 What does JPA provide? .............................................. 2
1.4 JPA providers .......................................................... 2

## 2 Persistence context

2.1 Overview .............................................................. 3
   2.1.1 What is a persistence context? .............................. 3
   2.1.2 What is an entity manager? ................................. 3
   2.1.3 Optimistic concurrency control ............................ 3
   2.1.4 Life span of entity manager ............................... 3
   2.1.5 Transaction management .................................. 4
   2.1.6 The scope of a persistence context ....................... 4
   2.1.7 How are entity managers obtained? ....................... 4
   2.1.8 Detaching objects from a persistence context .......... 5
2.2 Persistence context configuration .......................... 5
   2.2.1 Configuring the persistence context for a non-managed application .......................... 6
   2.2.2 Configuration of persistence context for managed applications .......................... 6

## 3 Entities

3.1 Simple entities ................................................... 6
   3.1.1 Declaring entities ......................................... 6
   3.1.2 Requirements for entities .................................. 7
   3.1.3 What is persisted? ......................................... 8
   3.1.4 Embeddable classes ....................................... 9
   3.1.5 Primary keys ................................................ 9
   3.1.6 Specifying column mappings ............................... 12
   3.1.7 Column constraints ...................................... 12
   3.1.8 Primitive collections and maps ......................... 12
3.2 Relationships ..................................................... 13
   3.2.1 Summary of UML relationships ............................ 13
   3.2.2 Composition relationships between entities .......... 15
   3.2.3 Relationship types ...................................... 15
   3.2.4 Fetching strategies ....................................... 18
   3.2.5 Specialization ........................................... 18
4 The Java Persistence Query Language (JPQL)

4.1 JPQL versus SQL .............................. 22
  4.1.1 Result collections in JPQL ................. 22
  4.1.2 Selecting entity attributes ................ 23

4.2 Statement types .............................. 23
  4.2.1 Elements of JPQL query statement .......... 23
  4.2.2 Elements of update and delete statements .. 23

4.3 Polymorphism ................................ 24

4.4 Navigating object graphs ...................... 24
  4.4.1 Simple paths ................................ 24
  4.4.2 Single-valued versus multi-valued paths ... 25

4.5 Specifying the source of a query .............. 26
  4.5.1 Selecting from multiple domains .......... 26
  4.5.2 Joins .................................... 26

4.6 Collapsing multi-valued paths into Single-valued paths .... 27

4.7 Constraining a result set via a WHERE clause ... 27
  4.7.1 Comparison operators ...................... 27
  4.7.2 Calculation and logical operators ......... 28
  4.7.3 Using collection variables in WHERE clauses 28

4.8 Constructing result objects ................... 28

4.9 Nested queries ................................ 29

4.10 Ordering .................................... 29

4.11 Grouping .................................... 29
  4.11.1 Positional parameters ................... 29
  4.11.2 Named parameters ....................... 30

5 Constructing and executing queries .......... 30
  5.1 Named queries ............................... 30

6 JPA converters ................................ 30
  6.1 Defining custom converters? ................. 30
  6.2 Applying custom converters .................. 31
  6.3 Default converters ........................... 31

7 Calling stored procedures ..................... 31

8 The criteria API ................................ 32
  8.1 Overview .................................... 32
  8.1.1 Benefits of using the criteria API for specifying queries ... 32
  8.1.2 Query tree ................................ 33

  8.2 Generating the JPA metamodel ............... 33
  8.2.1 Maven build declarations to generate a canonical metamodel 33
  8.2.2 Generated metamodel classes .............. 35

  8.3 Simple example ................................ 35
  8.4 Query operators ............................. 36
  8.5 Composite predicates ......................... 36
  8.6 Ordering .................................... 37
  8.7 Joins ...................................... 37
1 Overview

The Java Persistence API (JPA) is a specification of a Java API for accessing, persisting, and managing data persisted in a relational database or in any other database for which there is a JPA provider.

JPA is used to persist POJOs (Plain Old Java Objects). These classes do not have to implement any particular interface or subclass any particular class.

1.1 What is JPA?

JPA, the Java Persistence API aims to provide a standard persistence framework to be used when persisting entities to a database. Initially JPA provided a standard API for Java-based object-relational mappers which enabled you to

- abstract from
  - any specific O/R framework (e.g. Hibernate, EclipseLink, ...), and
  - from any specific relational database and the flavour of SQL used by that database,
- use an object cache to improve scalability and performance.

In the mean time one can abstract fully from the database technology used using JPA/JDO bridges to persist to object databases and specific JPA adapters for different NOSQL databases.

Even though JPA is commonly used from within enterprise applications which often run in enterprise application servers, JPA can also be used within Java-SE applications.

1.2 Why use JPA?

1. Decoupling/abstraction
   • from JPA-provider, Technology-specific query language and Persistence Technology
   • no vendor lock-in

2. Reduce code bulk
   • plumbing code removed (code only business logic)

3. Reduce errors & improve consistency
   • mapping complex object graphs onto relational databases is error-prone
   • database schemas and structure created from object graphs

4. Improve performance
   • through object caching

5. Maintainability & Portability
   • Through code reduction, decoupling and JPA being a widely supported public standard.

6. JPA Criteria
   • Have powerful framework for dynamic query construction
1.3 What does JPA provide?

- O/R mapping
- Persisting, removing, querying, updating
- Object caching
  - with eager and lazy data retrieval
- Value objects and merging
- Object-Oriented query language with mapping onto technology-specific query language
  - including pre-compiled queries
- Concurrency support
- Constraint validation support
- Custom converters
- Dynamic query construction
- Calling stored procedures

1.4 JPA providers

There is a wide range of JPA implementations to choose from. Widely used examples include EclipseLink, Hibernate, OpenJPA and DataNucleus. These implementations compete on non-functional attributes (e.g. performance) and different implementation may be more suitable for different systems.

In addition, JPA implementations may provide non-standard extensions. It is advisable that these are either avoided or at least ring fenced. Otherwise the portability and flexibility of the application is compromised.

2 Persistence context

2.1 Overview

Even though a persistence context os one of the central concepts of JPA, it is often not very well understood.

2.1.1 What is a persistence context?

A persistence context is a cache of objects whose persistence is managed. The cache maintains objects in memory which can be efficiently manipulated without every time having to consult the database.

The persistence context typically maintains a set of non-shared database connections. Only one instance with the same object identity exists within a persistence context.
2.1.2 What is an entity manager?

An entity manager is an entity resource manager which is associated with a persistence context. It maintains a cache for the persistence context and the life cycle of the entity instances contained within that persistence context. The entity manager interacts with the object-relational mapper and uses a connection pool to interact with the persistence provider (e.g. database).

The entity manager is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Entities which are managed by an Entity Manager will automatically propagate these changes to the database when a transaction is committed. Entities which have been detached can be merged back into managed state resulting in any modifications made outside the persistence context being ultimately persisted to database upon commit.

2.1.3 Optimistic concurrency control

For highly scalable systems one usually requires optimistic concurrency control with versioning. Version or timestamp checking is used to detect

- conflicting updates across transactions, and to
- prevent lost updates within a transaction.

The behaviour is really the same as in concurrent version control systems like subversion or git or subversion. Different persistence contexts obtain their own detached copy of the entities. Which ever persistence context commits first will merge their changes into the global persistence context and persist its domain to the database.

Subsequent entity managers who commit may encounter a conflict when they merge their changes back into the global context. If this is the case, an exception is thrown. Otherwise the changes made within that transaction are persisted through to the database.

2.1.4 Life span of entity manager

JPA supports two types of persistence contexts. Transaction-scoped persistence contexts are specific to a single transaction whilst extended persistence contexts span across transactions.

In the case of transaction-scoped persistence context, one will obtain (in a managed application) or have to create (in a non-managed application) a new persistence context per transaction. Any entities which have been enlisted within the cache will be detached at the end of the transaction and any changes made after detachment will no longer be propagated into the database.

Transaction-scoped persistence contexts do not support optimistic concurrency control. They are thus largely used in non-managed applications which usually do not have high concurrency demands.

Extended persistence contexts maintain a cache across transactions. They provide thus more efficient caching and support optimistic concurrency control. Extended transaction contexts are typically used in managed applications where the caching and optimistic concurrency control are important to achieve the required scalability.

2.1.5 Transaction management

The transaction type for a persistence context may be either RESOURCE_LOCAL or JTA. In the case of RESOURCE_LOCAL the transaction management is provided by JPA which typically delegates it to the local resource (e.g. database). In the case of JTA a transaction manager implementing the Java Transaction API is used. Such entity managers can enlist multiple resources within a
transaction. In managed environments transaction boundaries are usually managed by the application server deducing the relevant transaction boundaries from the more abstract transaction requirements annotations (e.g. requires, requires-new, ...).

RESOURCE_LOCAL is often used in non-managed applications where transaction control may be required only for resources from a single database. When using RESOURCE_LOCAL, you must use the entity transaction API to begin commit around every call to your entity manager:

The entity manager for a managed environment is provided by the application server. It will usually use JTA transaction management, allowing for multiple resources (e.g. databases, message queues, external systems, ...) to be enlisted within a transaction.

2.1.6 The scope of a persistence context

The scope of a persistence context is the domain of entities which are managed by it, i.e. the collection of entities managed by the persistence context.

The scope can be specified in one of the following ways:

1. A persistence unit may refer to a orm.xml file defining the entities and how they should be mapped onto a relational database. This is specified in a <mapping-file> element in the persistence.xml.

2. You can use one or more <jar-file> elements to specify that the entity classes in those jar files need to be included in the persistence context.

3. You can have a list of <class> elements listing the entity classes to be managed within the persistence context.

4. The annotated entities contained in the root of the persistence unit which is the jar file or directory, whose META-INF directory contains the persistence.xml file. This approach is the common approach when defining the persistence context for managed applications.

2.1.7 How are entity managers obtained?

Depending on whether one performs persistence from a managed or non-managed application, the entity manager is either provided/injected by the environment (i.e. by the application server) or needs to be created manually.

In a container managed environment the entity manager is provided by the container and is obtained either via dependency injection by annotating an EntityManager field or via a JNDI lookup.

To obtain a JTA based entity manager you need to annotate the entity manager field with a @PersistenceContext annotation.

To obtain a entity manager using RESOURCE_LOCAL JTA provider, you annotate the entity manager field with a @PersistenceUnit annotation.

In a JavaSE application, the entity manager is not injected from a container, but must be created explicitly. For this you will

- define the persistence context descriptor, persistence.xml in a META-INF directory (or construct the persistence context properties in code),

- instantiate a entity manager factory for your persistence context from the general persistence environment, providing the entity manager factory a suitable name, and

- obtain an entity manager for your persistence context from entity manager factory.
Generally you should only have a single entity manager per persistence context active at any time. **Note:** Calling `entityManagerFactory.createEntityManager()` twice results in two separate `EntityManager` instances and therefore two separate PersistenceContexts/Caches.

```java
1. EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName);
2. EntityManager em = emf.createEntityManager();
```

### 2.1.8 Detaching objects from a persistence context

If an object leaves the cache/persistence context, it is detached from it. The entity is then in a value object state. This will, for example, happen when an object is serialized (e.g. by being passed as parameter in a remote service request). An object will also become detached if it exists beyond the life span of the entity manager (and hence cache). In addition the entity can be manually detached via

```java
1. entityManager.detach(myEntity);
```

Any updates made to a detached object are not reflected in the object cache and are not propagated to the database upon transaction commit.

You cannot call request an entity manager to persist or remove a detached entity (value object). Once the value object has been re-attached to the persistence context via

```java
1. entityManager.merge(myEntity);
```

it can be persisted and removed again.

**Note:** Due to lazy loading, detached objects (e.g. serialized parameters) may not have all the information populated.

### 2.2 Persistence context configuration

The way in which a persistence context is configured depends on whether it is a managed or a non-managed application.

#### 2.2.1 Configuring the persistence context for a non-managed application

For non-managed Java applications one needs to specify the database, database driver and login credentials which should be used as well as the set of entity classes which should be managed by the persistence context. The latter can be specified as a list of classes, in a separate `orm.xml` file or by specifying the jar-file(s) which contains the entity classes. The latter is often the most convenient approach:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<jar-file>myEntities.jar</jar-file>
<persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <properties>
    <property name="eclipselink.target-database" value="DERBY"/>
    <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/myDB;create=true"/>
    <property name="javax.persistence.jdbc.user" value="myApp"/>
    <property name="javax.persistence.jdbc.password" value="myApp"/>
  </properties>
</persistence-unit>
</persistence>
```
2.2.2 Configuration of persistence context for managed applications

The persistence unit configuration for a managed environment is specified in the persistence.xml file contained in the META-INF directory. It typically uses JTA-based transactions and refers to a data source defined for the container. In addition it can specify some properties for the object-relational mapper:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" 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">
<persistence-unit name="myEnterpriseApp" transaction-type="JTA">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<jta-data-source>jdbc/myDataSource</jta-data-source>
<properties>
<property name="eclipselink.ddl-generation" value="create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

3 Entities

Entities are data classes which exist from when they are created up to the point where they are explicitly removed. They are persisted through to persistent storage (e.g. database) and may survive the life span of a session or application.

3.1 Simple entities

3.1.1 Declaring entities

An Entity is defined by annotating them with javax.persistence.Entity

```java
import javax.persistence.Entity;

@Entity
public class Account {
  ...
}
```

One may customize the persistence by specifying, for example, the database table to which the entity should be persisted via

```java
@Entity(name="ACCOUNTS")
public class Account {
  ...
}
```
3.1.2 Requirements for entities

Entities must satisfy a number of requirements:

- **Constructors:** Entities must have a public or protected default (no-argument) constructor. They may have other constructors. If the default constructor is declared protected, it is only available for the entity manager and users are forced to use the publicly available constructors.

- **Primary key:** Every entity requires a primary key, which may be a simple primary key represented by a bean field, or a composite key. The primary key is specified by annotating the relevant field with `javax.persistence.Id`.

- **Support for serialization:** Entities which are meant to be detachable in order to pass them around as value objects (i.e. sent to a client through a remote interface) must be serializable. These temporarily detached value objects can be re-attached to the entity manager at a later stage.

- **Final:** Neither the class, nor any of its methods, may be final. The JPA provider must be able to subclass your class, in order to provide natural interception points and to access protected fields not published via public access methods.

- **Entities and inheritance:** Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.

- **Entities may be abstract:** Shared data may be encapsulated in abstract entities which are subclassed by various concrete entities.

- **Fields accessed only via accessors or business methods:** Persistent instance fields must be declared with private, package or protected scope (preferably private), and can only be accessed directly by the entity class’s methods. Clients must access the entity’s state through accessor or business methods.

- **All annotations either at getter or field level:** The entity manager will access the fields either directly or via getters and setters. The method used depends on whether the annotations are don on the fields or on the getters. They should, however, not be mixed - i.e. they should be either all at the field level or all at the getter level. Alternatively the access type can be explicitly specified on an entity by annotating the entity with either `@Access(AccessType.FIELD)` or `@Access(AccessType.Property)`.

3.1.3 What is persisted?

The persistent state of an entity is defined by its fields. The fields are accessed by the entity manager either

- via accessors (getXXX() methods) following the JavaBeans specification, or

- via direct access to fields .

The schema used for a particular object is inferred, based on whether the primary key (the `@javax.persistence.Id` annotation) has been indicated on a field, or an accessor method.

For example, state management would be performed through the get/set service of the following class. The entity manager will call the services in order to extract the state to be stored in the database, or to populate an instance with information from the database.

The following are valid data types for persistent fields:
• Java primitives and primitives wrappers,

• The following built-in classes:
  – java.lang.String,
  – java.util.Date (requiring the @Temporal annotation),
  – java.math.BigDecimal and BigInteger,
  – java.sql.Time and Timestamp (requiring the @Temporal annotation),
  – byte[], Byte[], char[] and Character[],
  – java.sql.Blob and Clob,

• embedded classes,

• other entities,

• collections of primitives, and

• any other serializable objects.

Collection variables are automatically persisted. The supported java.util.Collection types are java.util.List, java.util.Set and java.util.Map.

Generics should be used (e.g. List<Account>), and the various relationship annotations (such as @OneToMany) are required to control the mapping, such as putting bi-directional mappings in place.

Collections are mapped onto separate tables. In the case of many to many relationships a join table is created.

Transient fields (fields which do should not be persisted, and hence do not form part of the object’s persistent state) are specified by

• in the case of property access annotating the getter or setter as @javax.persistence.Transient, or

• by declaring the field itself as transient using the Java language keyword, in the case of field access.

Field validation may be done in the setter methods, which may throw an exception. An exception will cause the controlling transaction (if any) to be rolled back.

**Note:**

• Consider using bean validation

• It is typically questionable whether use-case specific validation should be performed on the entity object at all: This should rather be enforced at services level (e.g. the session beans).

### 3.1.4 Embeddable classes

Entities may have as components finer grained objects which are persisted, not as separate entities, but are expanded as a set of columns within within the tables created for the entities within which they are embedded.

For example, a location may have a name, an address and geographic coordinates which include the degrees longitude and degrees latitude. Embedding the GeographicLocation class
within a `Location` entity would add the `degreesLongitude` and `degreesLatitude` columns to the `Location` table.

As such embedded objects have no persistent identity. Their identity is the role in the context of the owner entity.

**Note:** Embedded classes are only used for composition relationships between classes, i.e. no other object may obtain a reference to an embedded object.

An class which is meant to be embeddable within entity beans must be annotated as such using the `@Embeddable` annotation.

```java
@Embeddable
class GeographicLocation implements Serializable {
    // getters & setters
    private double degreesLongitude, degreesLatitude;
}
```

By default, the access type of an embeddable class is determined by the access type of the entity within which it is embedded. This can be changed by annotating the embeddable class with an `@Access` annotation whose value is either `AccessType.Field` or `AccessType.Property`.

**Note:** It is generally recommended to specify the access type of the embeddable explicitly in order to prevent potential object-relational mapping errors caused by the entity manager loosing track of the state due to access through both channels. This can happen when the embedded class is contained in an entity with one access type which is, in turn, part of an entity which uses another access type.

To embed an embeddable class within an entity one has to add a field for the embedded class and annotate it or the getter as `@Embedded`

```java
@Entity
class Location implements Serializable {
    // ...
    @Embedded
generic GeographicCoordinates getCoordinates() {
        return coordinates;
    }
    private String name;
    // ...
    private GeographicCoordinates coordinates;
}
```

### 3.1.5 Primary keys

For every entity one must specify a primary key which may be

- a simple primary key, or
- a composite primary key.

It is generally preferable to have a simple primary key which is independent of any business semantics.

Simple primary keys are persisted into a single database column which will be assigned a primary key constraint.

The following are valid data types for persistent fields:

- Java primitives and primitives wrappers, and
• java.lang.String

Although approximate numeric types like float or double are permitted, they should gen-
erally not be used due to their inability to represent absolute values.

A primary field is specified for an entity by annotating either

• an accessor method, or
• an instance field

with @javax.persistence.Id.

For example, the following code snippet specifies that the accountNo is to be used as a
primary key for accounts:

```java
@Entity
public class Account
{
    public int getAccountNo()
    {
        return accountNo;
    }

    @Id
    private int accountNo;
}
```

One will commonly request the entity manager/database to automatically generate the value
of a primary key (which will always be a unique value) by annotation the key with the @GeneratedValue
annotation:

```java
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Account implements Serializable
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long accountNumber;
}
```

The annotation takes parameters, which allows the developer to indicate the generator (such
as a particular database table), and/or to indicate the strategy to be used (typically realised by
the underlying database). When one “doesn’t care, as long as it is unique” the AUTO strategy
usually works well.

The JPA specification supports composite keys via primary key classes. A primary key class
is as an embeddable class whose properties form the primary key fields. It must have a
default constructor as well as setters and getters for the primary key fields.

```java
package za.co.solms.partsCatalog;

/**
 * Interface for a part identifier.
 */
public interface PartId
{
    public String getCode();
    public String getManufacturerId();
}
```

```java
package za.co.solms.partsCatalog;

public interface PartId
{
    public String getCode();
    public String getManufacturerId();
}
```

```java
package za.co.solms.partsCatalog;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Account implements Serializable
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long accountNumber;
}
```
3.1.6 Specifying column mappings

The object-relational mapping can be customized in the `orm.xml` entity descriptor file or via in-code annotations. For example, the column name, length and precision can specified via the `@Column` annotation:
3.1.7 Column constraints
Commonly column constraints include a specification on whether a column is required or not and whether the entries in the column need to be unique:

```java
@Entity(name = "VHCL")
public class Vehicle {
    @Column(name = "REG_NO", length = "10", nullable = false, unique = true)
    public String getRegistrationNumber()
    {
    ...
```}

3.1.8 Primitive collections and maps
Can annotate collections and maps of basic types as `@ElementCollection`. If the storage provider is a relational database, collections of primitives are mapped onto a separate table with a link column and value columns. Maps of primitives onto primitives are mapped onto a separate table with one link column, one key column and one value column.

The mapping can be customized using the `@CollectionTable` annotation which allows you to specify the name of the table. Maps can be additionally annotated with a `@MapKeyColumn(name="...")` annotation.

```java
@Entity
public class CarPriceList implements Serializable {
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "UnavailableCars")
    public List<String> withdrawnCars;

    @ElementCollection(fetch = FetchType.EAGER)
    public Map<String, double> activeCarPrices;
```}

3.2 Relationships
JPA supports persistent relationships which are mapped down to database layer. There is some limited support for the standard object-oriented relationships:

- association and aggregations,
- composition, and
- specialization.
3.2.1 Summary of UML relationships

Figure 1 summarizes the UML relationships. It shows that these are conceptually specializations of each other and that we have weak and strong variants of “is a”, “has a” and “uses.”

Instances of the one class, the user, make, at times, use of instances of the other class, the service provider. The latter is often modelled as an interface in order to decouple the user from any particular implementation of a service provider. For example, clients of the bank, upon spotting an ATM, may decide to use it in order to withdraw some cash from their account, but they do not maintain a message path to any particular ATM. A dependency is called a “weak uses” because the user does not maintain a message path and is not in a position to, at any stage, send further service requests to the service provider.

Association is used for two purposes. On the one side it is used purely for navigability. In the second case it is used for a client server relationship (or peer-to-peer in the case of binary associations). In either case, the object which has the association maintains a message path to the associated object. It is conceptually a special form of dependency where the client still, at times, makes use of the service provider, but now the client maintains a message path to the service provider. For example, an amplifier has a message path to the speakers (the cables) in order to send service requests to them.

An association is called a “strong uses” because the client maintains the relationship and is in a position to send, at any stage, further service requests to the service provider.

Aggregation is a special form of association. The aggregate object still maintains a message
path to the component. It still can make use of the components. For example, in the context of a portfolio calculating its value, it will request the value of each asset and sum them up. However, in aggregation a state transition in the component may imply a state transition in the aggregate object, i.e. aspects of the component state are part of the state of the aggregate object.

In our example, a change in the value of any of the assets results in a change in the value of the portfolio. Aggregation is a weak has a relationship because it does not take exclusive control of the component. The component can be accessed directly and may be part of other aggregate objects. Furthermore, the asset can survive the portfolio. For example, a particular asset may be part of a number of different portfolios. A change in its value results in the value of multiple portfolios changing. Furthermore, one may decide to remove a portfolio (a particular grouping view onto one’s assets), but the assets would still survive.

Composition is a special type of aggregation (and hence also a special type of association and a special type of dependency). If the component state changes, the state of the owner also changes. The owner also maintains the message path and may, at any stage, issue further service requests to the component. Now we have, however, a “strong has a” relationship where the owner takes full responsibility for the component and encapsulates the component.

If a user of the DVD player wants to send a service request to its laser, it will have to do so via the services offered by the DVD player itself. If the laser is broken, the DVD player is broken too (it is responsible for the laser). Finally, should we decide to scrap the DVD player, the laser will be scrapped also.

Realisation is a weak is a relationship. It is used to show that a service provider implements an interface (and often a complete contract). This facilitates substitutability of one service provider with any other realising the same contract.

Specialisation is a very strong relationship which should be used with care. It is commonly used for data or value objects. Specialisation can be conceptually seen as special form of realisation in that the sub-class is a specialised realisation of the super-class. One can say, specialisation inherits substitutability from realisation. It can also be seen as a special form of composition as every sub-class instance will create an encapsulated super-class instance through which it obtains the superclass attributes, services and relationships. The super-class instance for the sub-class cannot be accessed directly from outside the sub-class instance. It will also not survive the sub-class instance. The superclass instance is part of the state of the sub-class instance. If the state of the superclass instance changes, the state of the sub-class instance changes too.

For example, assume a home loan application inherits a loan amount from loan application. If the loan amount changes the state of the home loan application changes. The sub-class instance also maintains a message path to the super class instance (super in Java and base in C#). It is thus also a special for of association. It may, for example make use of a superclass service via super.serviceRequest().

Containment is a separate relationship where instances of one class can only exist in instances of another. There are examples of such relationships in nature.

In order to determine the correct relationship between two classes one can take a requirements driven approach - similar to a shopping list for relationships. In either case one should always choose the weakest relationship which fulfils one’s requirements. The process of determining the correct relationship goes along two legs. On the one side you are trying to establish the type of dependency between the two classes. On the other side you will assess the level of substitutability and inheritance required.

First we assess whether there is a dependency between the classes. If instances of one class, A, never make use of instances of another class, B, and if one also does not need to be able to navigate from an A to a B, then there is not much of a relationship between these classes. Otherwise there is at least a dependency of A on B.
Next ask yourself whether instances of A should maintain a message path to instances of B. If so, upgrade the dependency to an association. If not, leave the relationship as a dependency. If we reached this point, we have at least an association from A to B. Next you can ask yourself whether any change in the state of an instance of B results in a change of state in the instance of A which maintains an association to it. If the answer is yes, then upgrade the association to an aggregation relationship. Otherwise leave it as an association.

If we reached this point we have at least an aggregation relationship from A to B. Next, you can ask yourself whether any change in the state of an instance of B results in a change of state in the instance of A which maintains an association to it. If the answer is yes, then upgrade the association to an aggregation relationship. Otherwise leave it as an aggregation relationship.

**Note:** If you decided on composition, you can do the following test to check whether you perhaps made an error. Check whether it would make sense for the component to outlast (sur-vive) the owner. If the answer is yes, then the relationship could not have been a composition relationship.

Next let us look at the plug-ability requirements. If the class should be pluggable (i.e. if the service provider should be substitutable), then one should introduce a contract for the service requirements. In the bare form, the contract is simply an interface and we have a realisation relationship. In order to assess whether you should upgrade the realisation relationship to a specialisation relationship, assess whether you want to inherit common properties and services.

**Note:** In general we would recommend to favour interfaces and realisation above inheritance and specialisation. The latter tends to result in very rigid designs which are difficult to modify. One may choose to use specialisation only for value or data objects which do not perform significant functionality.

### 3.2.2 Composition relationships between entities

In a composition relationship the component may not survive the owner. This is supported in JPA via the cascading relationship attribute. Cascading is supported for create, merge and remove operations. **Note:** Cascading-delete enforces that the component entity bean is removed when the owner is removed.

### 3.2.3 Relationship types

JPA supports uni-directional and bi-directional one-to-one, one-to-many, many-to-one and many-to-many relationships.

For each relationship there is a relationship owner who maintains the pointer (e.g. foreign key) of the relationship. In the case of bi-directional relationships the related entity also provides a message path to the relationship owner.

Consider the uni-directional single-valued relationship shown in the following figure:

The relationship owner simply maintains the message path as well as the foreign key. The mapping onto entity beans would be as follows:

**Note:** Cascading is specified to request the composition behaviour, i.e. that the component should not outlast the owner.

Consider the bi-directional one-to-one relationship shown in the following figure:

Here both entities maintain message paths to one another. At database level, there is, however, only one foreign key maintained, i.e. only one relationship owner.

```java
@Entity
public class PurchaseOrder {

    // ... fields, getters, setters, constructors...
}
```
For the reverse relationship we specify a `mappedBy` attribute which ensures that this relationship is not implemented at storage level:

To implement one-to-many or many-to-many relationships we need to use either one of the `java.util` collection types:

The mapping onto entity beans would be as follows:
Figure 4: A bidirectional many-to-one relationship

```
4  public Collection<Account> getAccounts() { return accounts; }
5  ...
6  @OneToMany(mappedBy="client")
7  private Collection<Account> accounts;
8  }
9  ...
10 @Entity
11 public class Account
12 {
13  public Client getClient() { return client; }
14  public void setClient(Client c) { client = c; }
15  @ManyToOne
16  private Client client;
17  }
18 }
```

Depending on your design, you may desire that operations on a parent component (e.g. a Client) cascade to its constituent components (for example, deleting a client may or may not cause all the client’s accounts to be deleted). This is specified with the cascade parameter of any of the relationship annotations, with a set of enumerated values provided by the enumeration `javax.persistence.CascadeType`. For example, to cause all operation (including deletion) to be cascaded to the constituent component:

```
1  public class Client
2  {
3    ...
4    @OneToMany(cascade=CascadeType.ALL)
5    private Portfolio portfolio;
6  }
```

The allowable values are:

- **ALL**: Cascade all operations
- **MERGE**: Cascade merge (update) operation
- **PERSIST**: Cascade persist operation
- **REFRESH**: Cascade refresh operation
- **REMOVE**: Cascade remove operation
- **DETACH**: Cascade detach operations
From the perspective of mapping object-oriented relationships onto persistent storage, one should use not use any cascading for association and for composition one should use `CascadeType.All`.

### 3.2.4 Fetching strategies

Fetching strategies are used to optimize performance and scalability based on the expected usage of entities. They determine how much of an object graph is loaded into the cache when an entity is retrieved from persistent storage. The options are `EAGER` and `LAZY` fetching which respectively fetch or do not fetch the associated entity which has been annotated with the corresponding fetching strategy.

For example, below we request eager fetching of the client entity when retrieving an order entity, i.e. when the order is retrieved from the database, the associated client entity is also loaded into the cache:

```java
@Entity
public class Order {
    ...
    @ManyToOne(
        fetchType = FetchType.EAGER)
    public Client getClient() {
        ...
    }
}
```

The default fetching strategies in JPA are `EAGER` for one-to-one and many-to-one relationships and `LAZY` for one-to-many and many-to-many relationships.

### 3.2.5 Specialization

The concept of specialization and substitutability is not directly supported in relational database management systems. In order to support OO ↔ Relational mapping, we need to map specialization relationships onto relational databases.

JPA supports mapping of specialization relationships as well as polymorphism through to persistent storage level. To this end JPA supports a range of mapping strategies. None is perfect and JPA provides the option of requesting a mapping strategy which makes the appropriate quality attribute trade-offs for the problem at hand. In particular, one commonly trade-offs performance and scalability for maintainability and improved semantics.

JPA supports 4 types of mappings of specialization relationships onto relational databases:

1. Joined subclass
2. Single table per class hierarchy
3. Table per class
4. Mapped superclass

This is usually the preferred mapping. Each class in the specialization hierarchy is persisted in its own table. Subclass tables have a primary key column which acts as foreign key to the primary key column of the superclass, i.e. the object identity is preserved across all abstractions of an object.

The annotation specifying the mapping strategy is inherited, i.e. it need only be specified for the ultimate base class of the specialization hierarchy. For example
When choosing the Single Table Per Class Hierarchy strategy, each entire class is persisted in a single, typically sparsely populated table. The table has columns for the primary key, fields of all properties of all classes in the class hierarchy, and a discriminator column identifying the concrete class for that instance.

This strategy requires only a single lookup (no table joins), but a change to any classes in the hierarchy or the addition of another sub-class requires changing the table structure.

In this mapping strategy each concrete subclass is separated by its own stand-alone table which contains all fields (also the inherited fields) of the class as well as a primary key column. The mapping results in a non-normalized database structure. Modifying a superclass will require modifying all tables for all concrete subclasses. On the other hand, an entity lookup is a single table lookup.

At time don’t want a separate table for an abstract superclass. Neither do we want to use a single table for class hierarchy. Instead we would like to embed superclass fields in table of concrete subclass. In such cases one would consider using a MappedSuperclass. The abstract base class would not be annotated as an Entity as no table is created for it.

A MappedSuperclass is typically use if an abstract superclass has only very few fields one does not want the overheads of query on another small table. The resultant persistence mapping is not normalized. One effectively trades maintainability off for performance and scalability whilst retaining pluggability and polymorphism.

A MappedSuperclass is effectively an Embeddable class. The latter is used for composition relationships whilst the former is used for specialization relationships.

To request that the abstract base entity should be embedded annotated as @MappedSuperclass. In addition, one can still specify an inheritance strategy for class hierarchy.

Consider, for example, we could introduce the concept of a Chargeable as a mapped superclass. It holds only the object identifier and a reference to the income account. The Product class adds only a price and the Service class only introduces the concept of a service without adding anything. All three are abstract super classes which could potentially be implemented as @MappedSuperclass.
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.MappedSuperclass;
import javax.persistence.OneToOne;

@MappedSuperclass @Inheritance(strategy=InheritanceType.JOINED)
public abstract class Chargeable {
    public Chargeable() {}
    public Account getIncomeAccount() {return incomeAccount;}
    public void setIncomeAccount(Account incomeAccount) {
        this.incomeAccount = incomeAccount;
    }
    public long getCode() {return code;}
    public void setCode(long code) {this.code = code;}
    @OneToOne
    private Account incomeAccount;
    @Id @GeneratedValue
    private long code;
}

import javax.persistence.MappedSuperclass;
public class Service extends Chargeable {
    public Service() {}
}

Figure 5: Using mapped super classes for Chargeables, Products and Services
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public class Product extends Chargeable {
    public Product() {}
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    double price;
}

import javax.persistence.Entity;

@Entity
public class Consultation extends Service {
    public Consultation() {} 
    public double getDuration() { return duration; }
    public void setDuration(double duration) { this.duration = duration; }

    private double duration;
}

import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.OneToMany;

@Entity
public class BloodTest extends Service {
    public BloodTest() {}
    public Set<Disease> getDiseaseCheckList() {
        return diseaseCheckList;
    }

    private Set<Disease> diseaseCheckList;
}

4 The Java Persistence Query Language (JPQL)

The Java Persistence Query Language (JPQL) is a storage technology-neutral object-oriented query language enabling users to formulate queries across object graphs. As such the conceptual queries specified in JPQL are mapped onto the query language for the chosen persistence technology like the SQL for the relational database you’ve chosen or OQL for an object database.
4.1 JPQL versus SQL

The structure of a JPA query is in many ways similar to a traditional SQL query. It is generally of the form in which

- **SELECT**: specifies the type of objects or values to be selected which may be
  - an entity,
  - a value object or
  - a primitive data type
- **FROM**: specifies the domain to which the query applies and
- **WHERE**: specifies constraints which restrict the result collection.

For example, if we have an `Account` entity with a balance field, we can issue the following query:

```sql
SELECT a
FROM Account a
WHERE a.balance > 0
```

4.1.1 Result collections in JPQL

A core difference between JPQL and SQL is that the result collection in JPQL will be a collection of references to one of

- entities,
- other Java objects which are expanded within the same table (embedded classes),
- Java primitives,
- new instances of Java result objects whose fields are populated from the query,

while in SQL the result is a new conceptual table with column entries sourced potentially from different tables, i.e. it can contain elements from different tables and hence elements extracted from different entities.

Ultimately the result collection will be either a standard `java.util.Collection` or `java.util.Set`.

4.1.2 Selecting entity attributes

We can use the element access operator to select specific attributes of an entity. For example

```sql
SELECT a.balance
FROM Account a
```

returns a collection of all account balances. The result collection will be of the data type of the `balance` field in the `Account` class (e.g. an instance of a `Money` class or a `Double`).

In this case the `Object(..)` phrase is dropped. The JPQL specification requires that you wrap your result with an `Object()` phrase only in that case where a stand-alone variable is returned without navigating along a path.
4.2 Statement types

The JPQL is syntactically similar to the Standard Query Language (SQL) in that it supports 3 types of statements:

1. **Select statements:** Select statements are used to access selected data in persistent storage.
2. **Update statements:** Update statements are used to modify information maintained in persistent storage.
3. **Delete statements:** Delete statements are used to remove information currently held in persistence storage.

4.2.1 Elements of JPQL query statement

The elements of a select statement are:

- a **select clause** which determines the type of the objects or values returned (in JPQL the result set is always a collection of objects or values), where the objects are either retrieved entities which match the query or new objects whose fields were populated from the query,
- a **FROM clause** which constrains the domain from which the selection is done,
- an optional **WHERE clause** which may be used to constrain the collection of objects selected from that domain,
- an optional **GROUP BY clause** which enables one to group query results in terms of groups,
- an optional **HAVING clause** used in conjunction with the GROUP BY clause in order to filter over aggregated groups, and
- an optional **ORDER BY clause** enabling one to request an ordering from the returned result objects/values.

4.2.2 Elements of update and delete statements

The update and delete statements contain only the **UPDATE/DELETE** clause and an optional **WHERE** clause.

4.3 Polymorphism

JPQL statements are intrinsically polymorphic. All statement elements which apply to a target bean also apply to all its specializations. The result is often a polymorphic collection.

4.4 Navigating object graphs

In an object-oriented query language one queries along associations including aggregation, composition and specialization relationships as well as primitive fields.
4.4.1 Simple paths

Assume we have a Bond entity bean which has the structure shown in Figure 6. One of the strengths of JPQL is its ability to smoothly navigate across relationships, i.e. through object graphs. Consider, as an example, the UML diagram for a bond shown in Figure 6.

In JPQL one can traverse relationships in an object-oriented fashion using the Java element access operator. For example, we could specify the following SELECT statement to select all bond accounts:

```
SELECT b.bondAccount FROM Bond b
```

returns a collection of accounts, each of which is a bond account. The Bond entity bean must supply an abstract accessor method to query the related bond account.

The equivalent SQL statement would look something like this:

```
SELECT account from Account, Bond
WHERE Bond.bondAccount = Account.id
```

Our query may span multiple nodes like in:

```
SELECT bond.bondAccount.balance FROM Bond bond
```

4.4.2 Single-valued versus multi-valued paths

A single valued path is a path without any branching below the highest layer (i.e. the layer connected to the result objects). SELECT clauses and most WHERE clauses require a single-
valued path.

For example, all the queries discussed in the previous section use single valued paths in the SELECT statement and are hence valid JPQL statements. On the other hand, querying all the bond accounts of all the clients via

```
SELECT client.bonds.account FROM Client client -> INVALID
```

resembles a multi-valued path because `c` refers to a collection of clients each of which has a collection of bonds which each has an account.

As a second example, consider the UML diagram for a course schedule shown in figure 7. In a relational database this object graph could be represented as 4 tables, one for each entity.

![Figure 7: UML class diagram for presentations of courses for a course schedule](image)

If we wanted to extract all course names which are currently scheduled i.e. for which there exists a presentation), we could do this via the following SQL query:

```
SELECT Course.name from Course, Presentation WHERE Presentation.course = Course.id
```

To achieve the same in JPQL we can specify the following query:

```
SELECT p.course.name FROM Presentation p
```

This is a single-valued path and hence the query is valid. On the other hand, the query

```
SELECT p.presenters.course.name FROM Presentation p -> INVALID
```

is incorrect because we have, once again, a multi-valued path.
4.5 Specifying the source of a query

The FROM clause specifies and constrains the domain of the query by specifying the domain as a particular entity type.

4.5.1 Selecting from multiple domains

The FROM clause supports selecting from multiple domains delimited by commas.

For example, should you wish to find all election results which had a greater attendance than South Africa’s 1994 election you could specify the following query:

```sql
SELECT DISTINCT election
FROM Election election, Election election94
WHERE election.turnout > election94.turnout AND
election.country = 'South Africa' AND
election94.year = '1994'
```

4.5.2 Joins

An JOIN clause is used to combine two or more entities which have some common property.

Inner joins are used to select from an inclusion set obtained by a join condition over different entities. They can be specified implicitly via the Cartesian product or explicitly.

Implicit inner joins join multiple paths from multiple entities implicitly. For example, the following query performs an implicit inner join to determine those companies which are both customers and service providers:

```sql
SELECT DISTINCT c
FROM Customer c, ServiceProvider sp
WHERE c.companyRegistrationNo = sp.id
```

The following explicit INNER JOIN returns a collection of all bond accounts of clients living in Johannesburg:

```sql
SELECT bonds.account
FROM Client c INNER JOIN c.bonds bonds
WHERE c.address.city = 'Johannesburg'
```

Here the INNER JOIN can be abbreviated to JOIN (INNER is optional). This is equivalent to

```sql
SELECT bonds.account
FROM Client c IN (c.bonds) bonds
WHERE c.address.city = 'Johannesburg'
```

While an inner join retrieves only those objects which satisfy the join condition, an outer join does the same thing but with the addition of returning objects from the left collection for which there were no matching objects in the right collection.

For example, assume we have a one-to-zeroOrOne relationship between book and publisher, i.e a book may or may not be published by a publisher. Now assume you want to retrieve all book entities and load the publishers for those books which have publishers into the cache. We thus use an outer join to retrieve the set of entities where matching values in the join condition may be absent.

```sql
SELECT b FROM Book b LEFT JOIN b.publisher p WHERE p.address.country = 'South Africa'
```
gets all books, irrespective of whether they do or do not have publishers and also loads all those publishers of books who are in South Africa into the cache.

4.6 Collapsing multi-valued paths into Single-valued paths

SELECT clauses are restricted to single-valued paths. The same is largely true for WHERE clauses. So, how do we handle queries along multi-valued paths?

In JPQL this is done by defining collection variables via an IN clause. Consider, for example the invalid query

```sql
SELECT client.bonds.account FROM Client client
```

The correct form of this query in JPQL is

```sql
SELECT bonds.account FROM Client c, IN(c.bonds) bonds
```

Here the IN-clause defines a collection variable, bonds, which, for each client, resembles the client’s bonds.

In a similar way we can fix the following invalid JPQL statement

```sql
SELECT p.presenters.course.name FROM Presentation p
```

by defining a collection variable, ps, via an IN clause

```sql
SELECT ps.course.name FROM Presentation p, IN(p.presenters) ps
```

4.7 Constraining a result set via a WHERE clause

Analogous to SQL, EJB-QL uses a WHERE clause to restrict the elements returned in the result collection. For example, we can select only those courses to which one or more presenters have been allocated via

```sql
SELECT Object(c) FROM Course c
WHERE c.presenters NOT EMPTY
```

4.7.1 Comparison operators

JPQL supports a relatively extensive set of comparison operators which can be used in where clauses:

- `==`, `<`, `>`, `<=`, `>=`, `<>`
- `BETWEEN`, `LIKE`, `IN`, `IS NULL`, `EMPTY`, `MEMBER OF` which can all be inverted by combining them with a
- logical operators `AND`, `NOT`, `OR`.
- ...

For example
4.7.2 Calculation and logical operators

JPQL support arithmetic operators (+ - * /), calculation operators (MAX, MIN, SUM, MOD, AVG, COUNT, SQRT) and a range of string operators (LENGTH, LOCATE, SUBSTRING, UPPER, LOWER, CONCAT).

4.7.3 Using collection variables in WHERE clauses

We often have to define collection variables for multi-valued path constraints in WHERE clauses. For example

```sql
SELECT Object(c) FROM Course c WHERE c.prerequisites.name = 'Programming in Java'
```

is invalid because of the match against a multi-valued path, while

```sql
SELECT Object(c) FROM Course c, IN(c.prerequisites) p WHERE p.name = 'Programming in Java'
```

4.8 Constructing result objects

The result of a JPQL query is always single object or collection of objects. Instead of returning persisted objects (e.g. entities, embedded objects, ...), the query can construct a collection of new objects which have been populated from information obtained from entities.

For example, the following JPQL query

```sql
SELECT NEW za.co.academics.UniversityInfo (u.name, u.address, c.name c.registeredStudents)
FROM University u JOIN u.course c WHERE c.registeredStudents > 100
```

creates a list of result objects populated from university and course entities.

4.9 Nested queries

JPQL supports nested queries, i.e. queries which have sub-queries embedded within the conditional expression of a WHERE or HAVING clause.

For example, to select the best student on a course, one could use

```sql
SELECT s FROM student s where s.courseResults.average = (SELECT MAX(s.courseResults.average) from student s)
```

4.10 Ordering

One can use the ORDER BY clause followed by either ASC or DESC to request that the result set should be ordered in ascending or descending order of some field.
For example, to return a list of soccer stadiums which can seat at least 10000 spectators in
the order of the number of spectators they can accommodate, one can use

```sql
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY s.numSeats DESC
```

To refine the sort order, one can use multiple comma-separated sort criteria which will be
applied in the order in which they are defined:

```sql
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY s.numSeats DESC, s.age ASC
```

4.11 Grouping

The GROUP BY construct enables the aggregation of values according to a set of properties. The
HAVING construct enables conditions to be specified that further restrict the query result.

For example,

```sql
SELECT j.publisher, count(j.circulation) FROM journal j GROUP BY j.publisher
HAVING COUNT(j.circulation) > 100000
```

selects all journals with a circulation of more than 100000 Note: The expression which appears
in the GROUP BY clause must appear in the SELECT clause.

4.12 Query parameters

One may specify query inputs either as positional or as named parameters. The query input can
only be used in the WHERE clause or HAVING clause of a query.

4.12.1 Positional parameters

Positional parameters are specified with a question mark (?) prefix followed by an integer des-
ingnating the position of the parameter. Input parameters are automatically numbered, starting
from 1. The same parameter can be used multiple times within a the same query.

```sql
SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```

4.12.2 Named parameters

Named parameters are case sensitive and their identifier is prefixed by the ":" symbol.

For example

```sql
SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```

5 Constructing and executing queries

The entity manager provides an API for the persistence context and object-relational mapper.
It is used to construct and execute queries:
List<Product> products = entityManager.createQuery
        ("SELECT p FROM Product p WHERE p.description like :descr")
        .setParameter("descr", description)
        .setMaxResults(30)
        .setFirstResult(pageNo*30)
        .getResultList();

5.1 Named queries

Named queries are statically defined queries with predefined, unchangeable query strings. They are typically pre-compiled.

Named queries are defined via the @NamedQuery annotation. This could be in entities or in sessions beans.

@NamedQuery(name="bonds.getAllAbove",
query="select b from Bond b where b.balance &gt;= :amount")

Named queries are instantiated and executed via

Query query = entityManager.createNamedQuery("bonds.getAllAbove");
query.setParameter(0, new Double(500000));
List<Bond> list = (List<Bond>)query.getResultList();

6 JPA converters

At times the data type in the database differs from the data type used in Java entity objects. For example, one might use a boolean in the Java class and have a zero or one in the database column for that field. Another example is to convert a Java list to a comma-delimited string. Hence, upon accessing the persistent storage, we need to convert between the data type used in the database and the data type used in the Java class – i.e. we need custom converters.

6.1 Defining custom converters?

@Converter
public class BooleanToIntConverter implements AttributeConverter<Boolean, String>
{
    public int convertToDatabaseColumn(Boolean value)
    {
        if (value)
            return 1;
        else
            return 0;
    }
    public Boolean convertToEntityAttribute(int value)
    {
        return (value == 1);
    }
}

6.2 Applying custom converters
6.3 Default converters

At times a conversion should apply by default to all fields of a type. In that case one annotates
the converter with @Converter(autoApply=true):

7 Calling stored procedures

Similar to the specification of a JPA named query, we can also define a named query for a stored
procedure.

We can then ask the entity manager to create an instance of the named procedure query, register
the stored procedure parameter types and set the corresponding parameter values. Finally we execute the query and extract the result:
8 The criteria API

8.1 Overview

The JPA Criteria API provides an object-based API for defining queries across object graphs as an alternative to String-based queries whose syntax is only checked at deploy time. One thus creates query objects.

Due to the nature of constructing the queries in code through objects, they provide a simpler, more natural API for dynamic query construction within object-oriented code.

JPA queries were introduced with JPA 2.0

8.1.1 Benefits of using the criteria API for specifying queries

- **Simpler dynamic query construction:** The Criteria API provides a simpler way to dynamically and incrementally assemble a complex query allowing for reuse of query elements across queries. This is particularly true for complex queries.

- **Compile-time checking of queries:** JPQL queries are typically checked only at run-time. Criteria-based queries are validated at compile-time. Consider for example the following query:

```
String jpqlQuery = "select weatherReading from WeatherReading where weatherReading.temperature > 40";
Query query = em.createQuery(jpqlQuery);
List<WeatherReading> result = query.getResultList();
```

The syntax error in the above query is not glaringly obvious and would only be spotted at either deploy-time or run-time (hopefully in the context of unit testing). The correct query string is, of course,

```
String jpqlQuery = "select weatherReading from WeatherReading weatherReading where weatherReading.temperature > 40";
```

On the other hand, when using the Criteria API, syntactically incorrect queries will result in compiler errors which are typically already highlighted by the IDE during coding.

- **Type-safe queries:** String-based JPQL queries are intrinsically not type-safe and one generally suppresses type-safety warnings via the corresponding annotation. For example,

```
@SuppressWarnings("unchecked")
public List<WeatherReading> getAllWeatherReadings() {
  Query query = entityManager.createNamedQuery("findAllWeatherReadings");
  return query.getResultList();
}
```

Criteria-based queries can, on the other hand, be used in a type safe way.

8.1.2 Query tree

When using the criteria API to assemble a query, one assembles a query tree with a root node representing the starting (from) point for the query. The nodes of a query tree represent the semantic query elements such as

- **WHERE** clauses,
• GROUP BY or ORDER BY clauses,
• sub-queries,
• ...

8.2 Generating the JPA metamodel

Since JPA criteria queries are assembled from instances of classes which define the persistence metamodel, one needs to generate the metamodel classes from the entity definitions. This can be done by using a Java APT annotation pre-processor which generates the canonical metamodel classes. In the case of EclipseLink this is provided by org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor.

8.2.1 Maven build declarations to generate a canonical metamodel

We need to generate the metamodel classes. This should be done in the generate-sources lifecycle phase so that the generated classes are available in the compile phase.

In the maven build we need to include

• A dependency on some JPA implementation (the JavaEE API does, of course, not provide them),
• A dependency on the object-relational mapper used (e.g. EclipseLink ),
• Configuring the Maven compiler plugin to compile for Java 7 or later (supporting annotations processing),
• Configuring the Annotations processor plugin to execute the annotations processor used to generate the canonical model generation, e.g. Eclipse ’s CanonicalModelProcessor.
• The required Maven repositories and plugin repositories.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-xsd_4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    ... <repositories>
        <repository>
            <id>mvn-repo</id>
            <name>Maven repository</name>
            <url>http://repo1.maven.org/maven2/</url>
        </repository>
        <repository>
            <id>java.net</id>
            <url>http://download.java.net/maven/</url>
        </repository>
        <repository>
            <id>EclipseLink Repo</id>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            ...</pluginRepository>
    </pluginRepositories>
</project>
```
<dependencies>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.0.0</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>${eclipselink.version}</version>
  </dependency>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <compilerArguments>-Xsourcecompatibility:1.8 -Xtarget:1.8 -Xmodule</compilerArguments>
      </configuration>
    </plugin>
  </plugins>
</build>
8.2.2 Generated metamodel classes

The purpose of the metamodel classes is to provide an infrastructure through which queries can be assembled from an object tree. They provide singular and collection attributes used to navigate the object graph.

```java
package za.co.solms.weather;
import javax.annotation.Generated;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;
import za.co.solms.location.Location;
@StaticMetamodel(WeatherReading.class)
public class WeatherReading_
{
    public static volatile SingularAttribute<WeatherReading, Integer> id;
    public static volatile SingularAttribute<WeatherReading, Double> humidity;
    public static volatile SingularAttribute<WeatherReading, Date> dateTime;
    public static volatile SingularAttribute<WeatherReading, Location> location;
    public static volatile SingularAttribute<WeatherReading, Ambiance> ambiance;
    public static volatile SingularAttribute<WeatherReading, Double> temperature;
}
```

8.3 Simple example

In this section we show a simple criteria-based query in order to introduce some of the core concepts. The simple query will resolve all weather readings for a particular location.

```java
public List<WeatherReading> getWeatherReadingsForLocation(Location location)
{
    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
    CriteriaQuery<WeatherReading> criteriaQuery = criteriaBuilder.createQuery(WeatherReading.class);
    Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
    Predicate predicate = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
    criteriaQuery.where(predicate);
    TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

In the above listing we

- ask the entity manager for a criteria builder and use it to create a criteria query which yields weather readings,
- specify the root (FROM domain) of the query,
- create a predicate which affirms weather readings which have the required location and add a WHERE node to the query using that predicate, and
• create a typed JPQL query from the criteria query and execute it, returning the type-safe result list.

8.4 Query operators

The query builder provides a range of query operators including

• arithmetic operators like sum, diff, prod, quot, min, max, avg, abs, and sqrt,
• relational operators like gt, ge, lt, le, equal, like and notLike,
• logical operators like and, or, xor, not
• set and collection operators like count, countDistinct, isEmpty, between, isMember, isNotMember, exists(subQuery), any(subQuery), all(subQuery)
• sorting operators like asc, desc
• date/time operators like currentDate, currentTime, and currentTimeStamp,
• string operators like upper, lower, concat, substring, and trim,
• data conversion operators like toDouble, toInteger, toLong, toString,
• and some general testOperators like isNull, isNotNull, isTrue, and isFalse.

8.5 Composite predicates

The criteria builder can be used to create composite logical expressions from multiple predicates. For example, if one wants to find all weather readings for a particular location for which the temperature was above 40°C, it can be done as follows:

```java
1 Predicate atLocation = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
2 Predicate tempGe40 = criteriaBuilder.ge(weatherReading.get(WeatherReading_.temperature), 40);
3 CriteriaQuery<WeatherReading> criteriaQuery = criteriaBuilder.createQuery(WeatherReading.class);
4 criteriaQuery.where(atLocation);
5 criteriaQuery.orderBy(criteriaBuilder.desc(weatherReading.get(WeatherReading_.temperature)));
```

8.6 Ordering

To request ordering, we add a orderBy node to the query tree, supplying it an ascending or descending operator obtained from the criteria builder:

```java
1 CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
2 CriteriaQuery<WeatherReading> criteriaQuery = criteriaBuilder.createQuery(WeatherReading.class);
3 Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
4 criteriaQuery.where(atLocation);
5 criteriaQuery.orderBy(criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location));
6 criteriaQuery.orderBy(criteriaBuilder.desc(weatherReading.get(WeatherReading_.temperature)));
7 TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
8 return query.getResultList();
```
8.7 Joins

If queries are based on multiple entities one may need to use joins. For this purpose JPA introduces the `Join` and `SetJoin` classes, both of which are generic, taking the joining `from` and the binding type as template parameters.

For example, if we want to find all students who have one or more enrollments which are not canceled, we could use the following join:

```java
CriteriaQuery<Student> q = cb.createQuery(Student.class);
Root<Student> c = q.from(Student.class);
SetJoin<Student, Enrollment> o = c.join(Student_.enrollments);
Predicate p = cb.equal(o.get(Enrollment_.status), Status.Canceled).negate();
c.where(p);
```