www.DBTechNet.org

DBTechNet Tutorial on

ORM – Object Relational Mapping

author: Arvo Lipitsäinen

Dpt. of Business Information Technology, Haaga-Helia University of Applied Sciences, Helsinki, Finland

Disclaimers

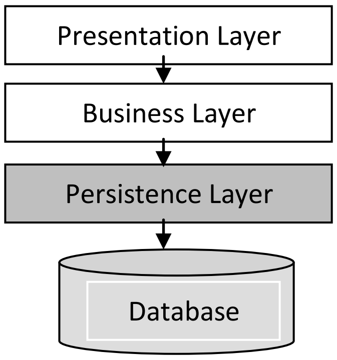
With the support of the EC LLP Transversal programme of the European Union.
This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author(s), and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Content

1. Layered software architecture

The object oriented paradigm uses a layered application software architecture, which shares the software logically into different layers. A typical, proven sharing criteria uses three layers:

1. Presentation layer
2. Business layer
3. Persistence layer



(Bauer, C. & King, G. 2007)

2. Object-oriented applications and Relational databases

Object persistence

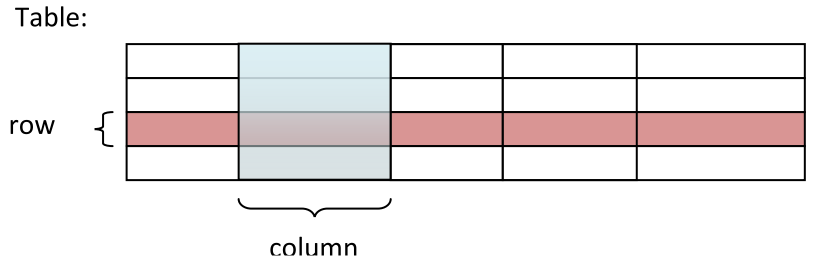
Object-oriented paradigm represents domain models as class diagrams and object models. In an object-oriented applications, persistence allows the state of an object to be stored to disk and the same state of the object can be re-created at some point in the future. This isn't limited only to single objects, many objects interconnected with each other can be made persistent and later re-created. (Bauer, C. & King, G. 2007)

Relational databases

Almost all applications need persistent data and nowadays the relational database management system (RDBMS) is the predominant technology for management of organization-wide persistent business data. RDBMS is based on the relational model introduced by E. F. Codd in 1970.

According to the relational model all data is represented as mathematical n-ary relations. A relation is defined as a set of n-tuples. A tuple is an unordered set of attribute values and an attribute is an ordered pair of attribute name and type name. The relational model allows the data be operated upon by means of relational operations of restriction, projection, Cartesian product and relational join. The relational model allows you to define data structures and constraints that guarantee the integrity of the data. (Bauer, C. & King, G. 2007)

Application programming interface of the relational database management system is SQL, hence the today's relational database products are called SQL database management systems or shortly SQL databases. SQL language references concepts of the relational model with different terms: a relation is a table, a tuple is a row in the table and the attribute is a column of the table. (Figure below)



Firstly, SQL databases are used from Java application with the Java Database Connectivity (JDBC) api.

Object Relational Mapping (ORM) means mapping of objects to tables in a relational database. Mapping is done using metadata describing relationships between object attributes and table columns. There are many ORM tools available in market e.g. Hibernate and Toplink. The tools enable at least basic database operations like create, read, update, and delete (CRUD).

The Java Persistence API (JPA) is the standard Java API for the management of persistence and object/relational mapping. The JPA is required to be supported in Java EE. It can also be used in Java SE environments.

Metadata for Object/Relational Mapping

The object/relational mapping metadata is part of the application domain model contract. It expresses requirements and expectations on the part of the application as to the mapping of the entities and relationships of the application domain to a database. Queries (and, in particular, SQL queries) written against the database schema that corresponds to the application domain model are dependent upon the mappings expressed by means of the object/relational mapping metadata.

It is permitted, but not required, that DDL generation be supported by an implementation of this specification. Portable applications should not rely upon the use of DDL generation.

3. Object/relational mapping paradigm mismatch

Bauer & King (Bauer, C. & King, G. 2007) introduce a list of object/relational mismatch problems:

1. **Problem of granularity**
   Java developers can freely define new classes in different levels of granularity. The finer-grained classes, like Address class, can be embedded into coarse-grained classes, like User class. Instead the type system of the SQL database are limited and the granularity can be implement only on two level: on table (User table) and column level (Address column).

2. **Problem of subtypes**
   The inheritance and polymorphism are basic and principal features of the Java programming language. New subclasses can be extended from a superclass so, that the data and method members of the superclass are visible in the subclasses and inherited methods can be overloaded or new methods and data members can be added in the subclasses. The polymorphism allows the members of subclasses to appear as and to be used like the members of the superclass. The inharitance of the Java language is on the theoretical level type inharitance. The SQL database products in generally

don't support type or table inharitance and if they support it, they don't follow standard syntax. The SQL databases also lack an standardized way to present polymorphic assosiations.

3. **Problem of identity**
   Java-objects define two different notions of sameness:
   - Object identity (roughly equivalent to memory location, checked with a==b).
   - Equality as determined by the implementation of the equals() method.

   The identity of a database row is expressed as the primary key. Neither equals() nor == is naturally equivalent to the primary key. In the context of persistence, identity is closely related to how the system handles caching and transactions.

4. **Problems relating to associations**
   In the domain model, associations represent the relationships between entities and associations are used for navigation. The Java language represents associations using object references, but the assosiations in relational databases are represented through the use of foreign keys.

   Directionality joins inherently to object referencies. The associations between objects are unidirectional pointers. When both end of the relationship has references to each other, the association is bi-directional. All associations in a relational database are effectively bi-directional, because the foreign keys are used to join tables and for projection,

   The multiplicity of the unidirectional association is possible to determine by looking only on Java classes. Java associations can have many-to-many multiplicity, table associations, on the other hand, are always one-to-many or one-to-one. So many-to-many association can be inplemented in the realtional database by using a new table for many-to-many association.

5. **Problem of data navigation**
   Accessing data and navigating from one object to another is different in Java and in the relational database. Accessing data in Java happens by navigating between objects through getter methods, e.g. order.GetLineItem().getProduct(). Accessing data from the relational database and navigating from one table to another with SQL requires joins between tables and performance of the queries can be improve by minimizing the number of the request to the database.

4. Entity

An entity is a lightweight persistent domain object. Typically an entity class represent a table in a relational database and each entity instance corresponds to a row in that table.

The entity class must be annotated with the `Entity` annotation. The entity class must be a top-level class and must not be `final`. No methods or persistent instance variables of the entity class may be `final`. The entity class must have a public or protected no-arg constructor. The entity class may have other constructors as well.

Every entity must have a primary key, which identifies the entity unambiguously. . The primary key is either a simple or composite primary key. A simple primary key must correspond to a single persistent field or property of the entity class. The `Id` annotation must be used to denote a simple primary key.

The `Table` annotation specifies the primary table for the annotated entity. If no `Table` annotation is specified for an entity class, the default name of the table is entity name. The `Column` annotation specifies a mapped column of the relational table for a persistent property or field. Default name of the column is the property or field name.

These annotations and types are in the package `javax.persistence`. XML metadata may be used as an alternative to these annotations, or to override or augment annotations.

Entity example

Below is an entity class, `Customer`, which is mapped to the `CUSTOMER` table with the `Table` annotation. Property `id` is the primary key, that is denoted with `Id` annotation before `getId()` method. The persistent fields or properties are not annotated, so they are mapped to the corresponding columns of the relational table.

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

@Entity
@Table(name = "CUSTOMER")
public class Customer implements Serializable{

  private int id;
  private String name;
  private String address;

  @Id
  public int getId() {
      return id;
  }
  public void setId(int id) {
      this.id = id;
  }
  public String getName() {
      return name;
  }
  public void setName(String name) {
      this.name = name;
  }
  public String getAddress() {
      return address;
  }
  public void setAddress(String address) {
      this.address = address;
  }
  public Customer(){}
  public Customer(int id, String name, String address){
      this.id = id;
      this.name = name;
      this.address = address;
  }
}
```

The persistent state of an entity is represented either by persistent fields or by persistent properties. The persistent fields are entity's instance variables and the persistent properties are JavaBeans-style properties which are accessed by getter and setter methods. Instance variables must not be accessed directly by clients of the entity, but they must be accessed only from within the methods of the entity itself. So the state of the entity is available to clients only through the entity's methods—i.e., accessor methods (get/set - methods) or other business methods.

The instance variables of a class must have `private`, `protected` or package visibility independent of whether field access or property access is used. When property access is used, the property accessor methods must be `public` or `protected`.

It is required that the entity class follows the method signature conventions for JavaBeans read/write properties for persistent properties when property access is used. For single-valued persistent properties, these method signatures are:

- `T getProperty()`
- `void setProperty(T t)`

Collection-valued persistent fields and properties must be defined in terms of one of the following collection-
valued interfaces regardless of whether the entity class otherwise adheres to the JavaBeans method conventions noted above and whether field or property access is used: `java.util.Collection`, `java.util.Set`, `java.util.List`, `java.util.Map`.

If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the `Serializable` interface.

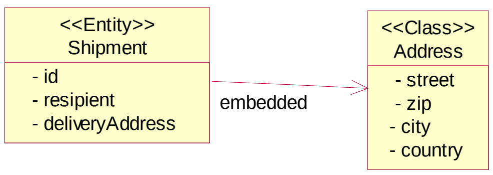
4.1 Embedded classes

An entity may use other fine-grained classes to represent entity state. Instances of these classes, unlike entity instances, do not have persistent identity of their own. Instead, they exist only as part of the state of the entity to which they belong. Embedded objects belong strictly to their owning entity, and are not sharable across persistent entities.

Embeddable classes must adhere to the requirements specified above for entities with the exception that embeddable classes are not annotated as `Entity`. Embeddable classes must be annotated as `Embeddable`.

The entity class specifies a a persistent field or property, whose value is an instance of an embeddable class, with the `Empedded` annotation.

Example.
The embeddable class `Address` is embedded into the `Shipment` entity.



Java code is below. The `deliveryAddress` field of the `Shipment` entity is annotated with `@Embedded` and the embebbable class `Address` is annotated with `@Embeddable`.

```
@Entity
public class Shipment
{
     private int id;
     private String recipient;
     private String date;
     @Embedded
     private Address deliveryaddress;
     private Order order;
 .
 .
}



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

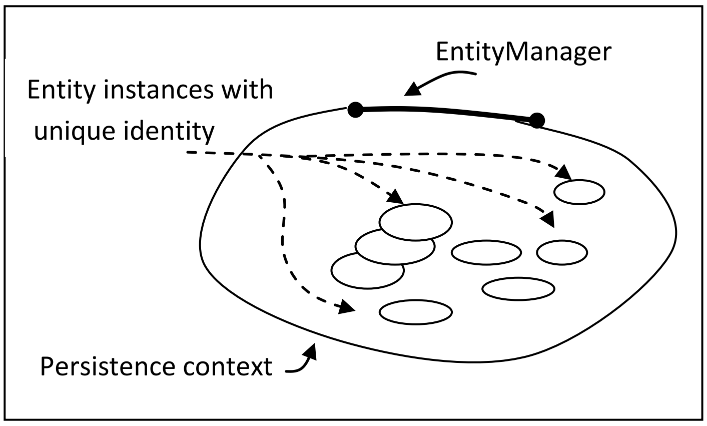
5. Managing entities

5.1 Persistence context

A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their life cycle is managed by a particular entity manager. An `EntityManager` instance is associated with a persistence context. The scope of the persistence context can either be the transaction, or can extend beyond lifetime of a single transaction.

A Persistence context can be managed by the container or by the application.

By default, the lifetime of the container-managed persistence context has transaction scope, so the persistence context ends when the transaction is committed or rolls back.



5.2 Entity Manager

`EntityManager API` is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities. The main methods of the `EntityManager` are below:

```
public interface EntityManager
{
   void close();
   Query createNamedQuery(String name);
   Query createNativeQuery(String sqlString);
   Query createQuery(String qlString);
   <T> T find(Class<T> entityClass, Object primaryKey);
   void flush();
   <T> T merge(T entity);
   void persist(Object entity);
   void refresh(Object entity);
   void remove(Object entity);
 }
```

In the table below are the main methods of the `EntityManager` with descriptions.

Method of `EntityManager`	Description

`void close()`	Close an application-managed entity manager.
`<T> T find(Class<T> entityClass, Object primaryKey)`	Find by primary key.
`void flush()`	Synchron ize the persistence context to the underlying database.
`<T> T merge(T entity)`	Merge the state of the given entity into the current persistence context.
`void persist(Object entity)`	Make an instance managed and persistent.
`void remove(O bject entity)`	Remove the entity instance.
`void refresh(O bject entity)`	Refresh the state of the instance from the database, overwriting changes made to the entity, if any.
`Query createQuery(String qlString)`	Create an instance of Query for executing a Java Persistence query language statement.
`Query createNativeQuery(String sqlString)`	Create an instance of Query for executing a native SQL statement,
`Query createNamedQuery(String name)`	Create an instance of Query for executing a named query (in the Java Persistence query language or in native SQL).

5.3 Persistence unit

The set of entities that can be managed by a given `EntityManager` instance is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be collocated in their mapping to a single database.

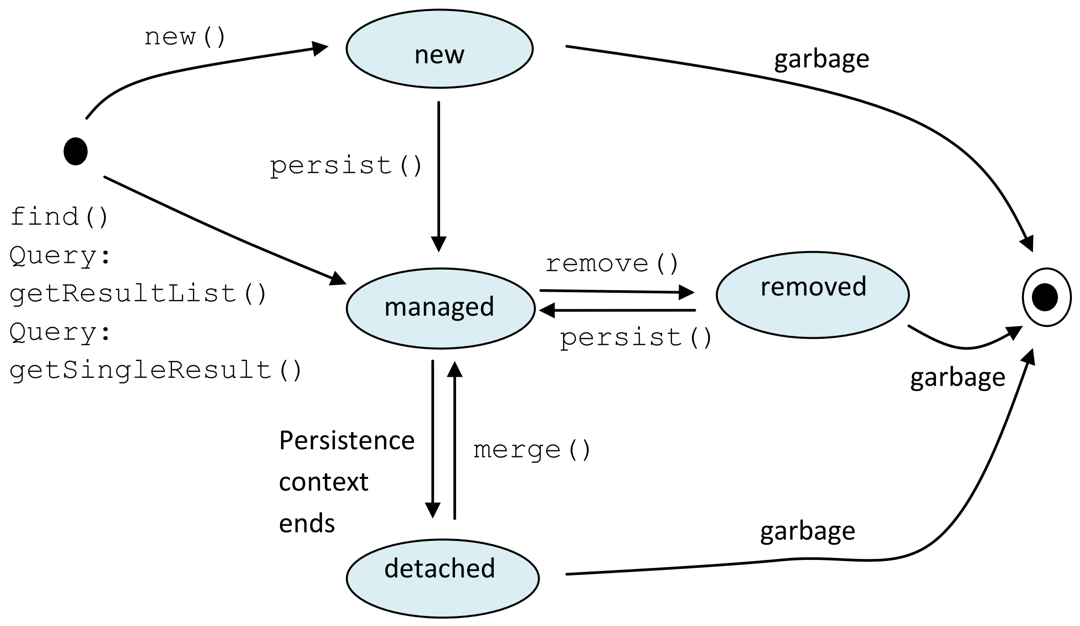
A persistence unit is defined by `persistence.xml` file, that resides in the `META-INF` directory.

```
<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="rvvtest" transaction-type="RESOURCE_LOCAL">
      <properties>
          <!-- Scan for annotated classes and Hibernate mapping XML files -->
          <property name="hibernate.archive.autodetection" value="class, hbm"/>

          <!-- DB2 ExpressC V9.x -->
          <property name="hibernate.connection.driver_class"
                    value="com.ibm.db2.jcc.DB2Driver"/>
          <property name="hibernate.connection.url"
                    value="jdbc:db2://localhost:50000/TEST"/>
          <property name="hibernate.connection.username"
                    value="RVV"/>
          <property name="hibernate.connection.password"
                    value="test"/>
          <property name="hibernate.dialect"
                    value="org.hibernate.dialect.DB2Dialect"/>

          <!-- JDBC connection pool (use the built-in) -->
          <property name="connection.pool_size" value="2"/>
      </properties>
   </persistence-unit>
</persistence>
```

5.4 States of entity instances

Object/relational mapping solutions don't only shield the developers from the details of the underlying database management system, but they also offer state management of the entity objects.

The figure below describes states of an entity instance's life cycle.



The entity instance can have following states:

- **New**
  After instantiating an entity with `new()` –operator the entity instance is in the new state. It has no persistent identity and is not yet associated with a persistent context and database. The state of the new instance is lost by garbage collection, when it has no referenced by any other object. The `persist()` method of entity manager transfer the entity instance from the new state to the managed state.

- **Managed**
  A managed entity instance has a persistent identity and is currently associated with a persistence context. It has a representation in the database. A managed instance has a primary key value set as its database identifier. When the persistence context ends, the state of the instance transfers from managed to detached.

- **Detached**
  A detached entity instance is an instance with a persistent identity but the instance is not (or no

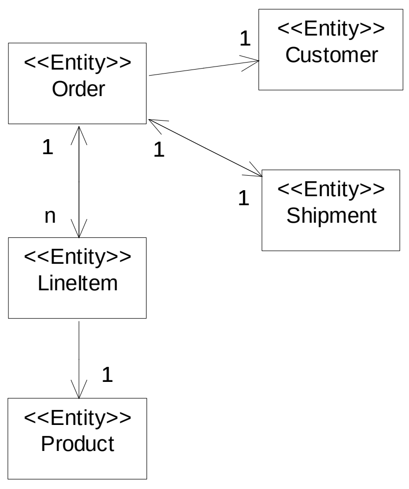
longer) associated with a persistent context. Its state is no longer guaranteed to be synchronized with the database state. A detached instance continues to live outside of the persistent context and its content can be modified in the program. The `merge()` method of the entity manager brings the detached instance back into the managed state.

- **Removed**
  The entity instance can be deleted by calling `remove()` method of the entity manager, which transfers the state from managed to removed. A removed entity instance is an instance with a persistent identity, associated with a persistence context, that will be removed from the database upon transaction commit.

6. Entity Relationships

The object-oriented domain model consists of business entities and relationships between entities. Relationships among entities may be one-to-one, one-to-many, many-to-one, or many-to-many. In addition, the relationships have a direction: the relationships can be unidirectional or bidirectional.

The figure below describes some business entities and relationships: `Order` entity has one-to-one relationships to `Customer` entity and to `Shipment` entity. The relationship between `Order` and `Customer` is unidirectional whereas relationship between `Order` and `Shipment` is birectional. In addition, the `Order` entity has bidrectional one-to-many relationship to `LineItem` entity and the `LineItem` entity has unidirectional one-to-one relationship to `Product` entity.



Relationships must have an owning side. An unidirectional relationship has only an owning side. A bidirectional relationship has both an owning side and an inverse (non-owning) side. The owning side of a relationship determines the updates to the relationship in the database.

If there is an association between two entities, one of the following relationship modeling annotations must be applied to the corresponding persistent property or field of the referencing entity: `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany`.

Relations Mapping Defaults

6.1 One-to-one relationship

An one-to-one relationship can be unidirectional or bidirectional. The unidirectional relationship is represented in section 6.1.1 and the bidirectional one in section 6.1.2.

6.1.1 Unidirectional one-to-one relationships

Assuming that:

Entity A references a single instance of Entity B.
Entity B does not reference Entity A.

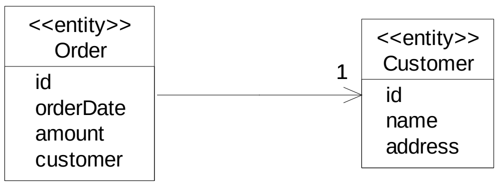
A unidirectional relationship has only an owning side, which in this case must be Entity A.

The following mapping defaults apply:

Entity A is mapped to a table named `A`.
Entity B is mapped to a table named `B`.

Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B` and there is a unique key constraint on it.

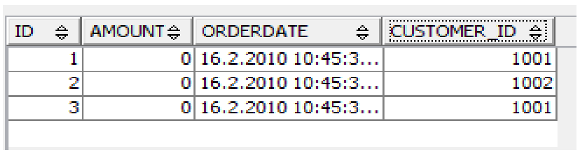
Example
There are `Order` and `Customer` entities and an unidirectional one-to-one relationship between them (figure below). The `id`, `orderDate` and `amount` fields of the `Order` entity and all fields (`id`, `name`, `address`) of the `Customer` entity are persistent fields. In addition, the `Order` instance has a `customer` relationship field/property annotated with the `OneToOne` annotation, which references to a `Customer` instance. Because the relationship is unidirectional, the `Customer` instance has no reference to the `Order` instance. See the Java codes of the both entities below.



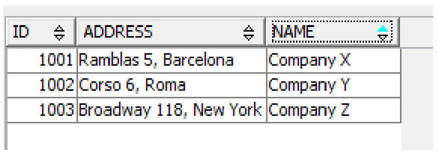
Mapping of the entities to the relational model can be seen in the figure below. The `Order` entity of the object model is mapped to the `ORDERHH` table of the relational model. The `Customer` entity is mapped to `CUSTOMER` table. The persistent properties of the both classes are mapped to the default columns of the tables.

Notice the mapping of the relationship to the foreign key column (`CUSTOMER_ID`) of the owning side of the relationship (`ORDERHH` table).

ORDERHH table **CUSTOMER table**



ID	AMOUNT	ORDERDATE	CUSTOMER_ID
1	0	16.2.2010 10:45:3...	1001
2	0	16.2.2010 10:45:3...	1002
3	0	16.2.2010 10:45:3...	1001



ID	ADDRESS	NAME
1001	Ramblas 5, Barcelona	Company X
1002	Corso 6, Roma	Company Y
1003	Broadway 118, New York	Company Z

The Java program codes are below.

The `Order` entity is mapped to `ORDERHH` table. (`Order` is a reserved term in the SQL and cannot be the name of a relational table.) The `id` property is annotated to the primary key by adding the `Id` annotation before the `getId()` method. With the `Id` annotation can be a `GeneratedValue` annotation, which specifies a generation strategy for the values of the primary key. `GenerationType.Auto` indicates that the persistence provider should pick an appropriate strategy for the particular database. See the `OneToOne` annotation before the `getCustom()` method (= `custom` property).

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

@Entity
@Table(name = "ORDERHH")
public class Order  implements Serializable {

     private int id;
     private String orderDate;
     private double amount; /* euros */
     private Customer customer;

     @Id @GeneratedValue(strategy=GenerationType.AUTO)
     public int getId() {
        return id;
     }
     public void setId(int id) {
        this.id = id;
     }
     public Date getOrderDate() {
        return orderDate;
     }
     public void setOrderDate(Date orderDate) {
        this.orderDate = orderDate;
     }
     public double getAmount() {
        return amount;
     }
     public void setAmount(double amount) {
        this.amount = amount;
     }
     @OneToOne
     public Customer getCustomer() {
        return customer;
     }
     public void setCustomer(Customer customer) {
        this.customer = customer;
     }
     public Order() {
        orderDate = new java.util.Date();
```

```
         amount = 0.0;
      }
}
```

The `Customer` class is below. The `Id` annotation can be before the id field or `id` property (`getId()`).

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

@Entity
@Table(name = "CUSTOMER")
public class Customer implements Serializable {

      @Id
      private int id;
      private String name;
      private String address;

      public int getId() {
         return id;
      }
      public void setId(int id) {
         this.id = id;
      }
      public String getName() {
         return name;
      }
      public void setName(String name) {
         this.name = name;
      }
      public String getAddress() {
         return address;
      }
      public void setAddress(String address) {
         this.address = address;
      }
      public Customer(){}
      public Customer(int id, String name, String address){
         this.id = id;
         this.name = name;
         this.address = address;
      }
}
```

The following `ManageOrder` program creates some `Customer` and `Order` instances with relationship between them. In addition the program make some queries to `Order` and `Customer` instancies. The query language is represented in the chapter 7.

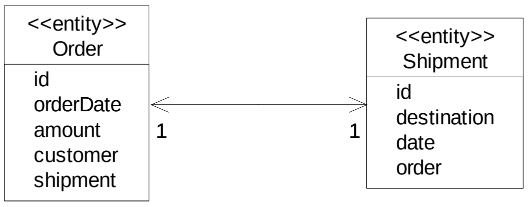
```
import javax.persistence.*;

import org.dbtech.entity.Customer;
import org.dbtech.entity.Order;

public class ManageOrder
{
    @PersistenceContext
    private EntityManager em;

    public void createCustomers(){

       Customer   customer = null;
       customer = new Customer(1001, "Company X", "Ramblas 5, Barcelona");
       em.persist(customer);
       customer = new Customer(1002, "Company Y", "Corso 6, Roma");
       em.persist(customer);
       customer = new Customer(1003, "Company Z", "Broadway 118, New York");
       em.persist(customer);
    }

    public boolean createOrder(int customerId){
       Customer customer = em.find(Customer.class, customerId);
       if (customer == null) return false;
       Order order = new Order();
       order.setCustomer(customer);
       em.persist(order);
       return true;
    }

   public List<Order> listAllOrders(){
       Query q = em.createQuery("SELECT o FROM Order o");
       return (List<Order>)q.getResultList();
   }

   public List<Order> listOrdersOfCustomer(int customerId){
       Query q = null;
       q = em.createQuery("SELECT o FROM Order AS o JOIN o.customer AS c
                                                        WHERE c.id = ?1");
       q.setParameter(1, customerId);
       return (List<Order>)q.getResultList();
   }
}
```

6.1.2 Bidirectional one-to-one relationship

Objects with bidirectional relations will be mapped to the relational model in the same way than in the unidirectional case.

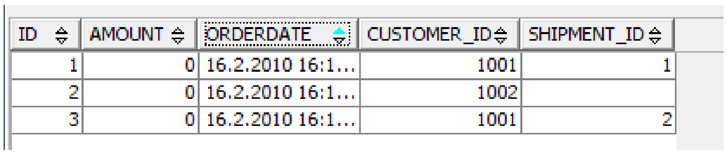
Example

`Order` and `Shipment` entities have a bidirectional on-to-one relationship (figure below). The `Order` entity has `shipment` relationship field/property annotated with `OneToOne`, which refers to `Shipment` instance and the `Shipment` entity has `order` relationship field/property annotated with `OneToOne`, which refer to `Order` instance.



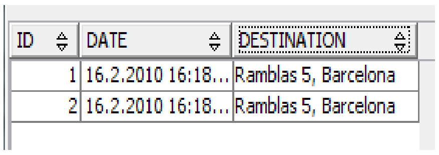
The bidirectional relationships must have a owner side. `Order` class is the owner side of this example. Thus the relationship is mapped to the `SHIPMENT_ID` foreign key of the `Orderhh` table.

ORDERHH table



ID	AMOUNT	ORDERDATE	CUSTOMER_ID	SHIPMENT_ID
1	0	16.2.2010 16:1...	1001	1
2	0	16.2.2010 16:1...	1002	
3	0	16.2.2010 16:1...	1001	2

SHIPMENT table



ID	DATE	DESTINATION
1	16.2.2010 16:18...	Ramblas 5, Barcelona
2	16.2.2010 16:18...	Ramblas 5, Barcelona

Java code of the `Order` and `Shipment` classes are below. Notice `OneToOne` annotation of the `order` relationship property of `Shipment` class has `mappedBy` element, that indicates the field of the owning side that owns the relationship. This element is only specified on the inverse (non-owning) side of the relationship.

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

@Entity
@Table(name = "ORDERHH")
public class Order  implements Serializable {

    private int id;
    private String orderDate;
    private double amount; /* euros */
    private Customer customer;
    private Shipment shipment;

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public Date getOrderDate() {
        return orderDate;
    }
    public void setOrderDate(Date orderDate) {
        this.orderDate = orderDate;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    @OneToOne
    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
    @OneToOne
    public Shipment getShipment() {
        return shipment;
    }
    public void setShipment(Shipment shipment) {
        this.shipment = shipment;
    }
    public Order() {
        orderDate = new java.util.Date();
        amount = 0.0;
    }
}
```

The Java code of the `Shipment` class is following.

```
import javax.persistence.*;

@Entity
public class Shipment {
    private int id;
    private String destination;
    private String date;
    private Order order;

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getDestination() {
        return destination;
    }
    public void setDestination(String destination) {
        this.destination = destination;
    }
    public String getDate() {
        return date;
    }
    public void setDate(String date) {
        this.date = date;
    }
    @OneToOne(mappedBy="shipment")
    public Order getOrder() {
        return order;
    }
    public void setOrder(Order order) {
        this.order = order;
    }
}
```

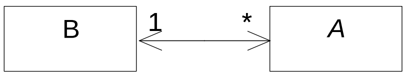
6.2 One-to-many/many-to-one relationships

Also one-to-many relationships can be bidirectional (6.2.1) or onedirectional (6.2.2).

6.2.1 Bidirectional many-to- one/one-to-many relationship

Assuming that:

Entity A references a single instance of Entity B.
Entity B references a collection of Entity A.

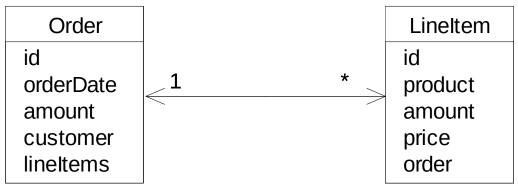


Entity A must be the owner of the relationship.

The following mapping defaults apply:

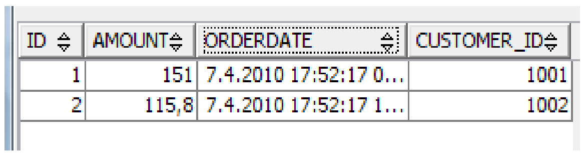
Entity A is mapped to a table named A.
Entity B is mapped to a table named B.
Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B.

Example.
Order and LineItem entities have a bidirectional one-to-many relationship (figure below).The LineItem class is the owner of the relationship.



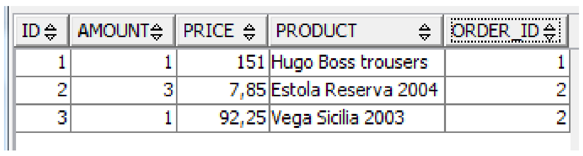
The Order class is mapped to the ORDERHH table and the LineItem class is mapped to the LINEITEM table. The relationship is mapped to Order_Id column of the LINEITEM table.

ORDERHH table



ID	AMOUNT	ORDERDATE	CUSTOMER_ID
1	151	7.4.2010 17:52:17 0...	1001
2	115,8	7.4.2010 17:52:17 1...	1002

LINEITEM table



ID	AMOUNT	PRICE	PRODUCT	ORDER_ID
1	1	151	Hugo Boss trousers	1
2	3	7,85	Estola Reserva 2004	2
3	1	92,25	Vega Sicilia 2003	2

Java code of Order and LineItem classes are below. The Order class has the Collection<LineItem> type lineItems relationship property with OneToMany annotation with the mappedBy element, that indicates the field of the owning side that owns the relationship. This element is only specified on the inverse (non-owning) side of the relationship.

```
import java.io.Serializable;
import java.util.Collection;
import java.util.Date;
import javax.persistence.*;
```

```
@Entity
@Table(name = "ORDERHH")
public class Order  implements Serializable {
        private int id;
        private String orderDate;
        private double amount; /* euros */
        private Customer customer;
        private Collection<LineItem> lineItems;

        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        public int getId() {
          return id;
        }
        public void setId(int id) {
          this.id = id;
        }
        public String getOrderDate() {
          return orderDate;
        }
        public void setOrderDate(String orderDate) {
          this.orderDate = orderDate;
        }
        public double getAmount() {
          return amount;
        }
        public void setAmount(double amount) {
          this.amount = amount;
        }
        @OneToOne
        public Customer getCustomer() {
          return customer;
        }
        public void setCustomer(Customer customer) {
          this.customer = customer;
        }
       @OneToMany(cascade=CascadeType.ALL,fetch=FetchType.EAGER,mappedBy="order")
        public Collection<LineItem> getLineItems() {
          return lineItems;
        }
        public void setLineItems(Collection<LineItem> lineItems) {
          this.lineItems = lineItems;
        }
        public void addLineItem(LineItem lineitem){
          lineItems.add(lineitem);
        }
        public Order() {
          orderDate = new Date();
          amount = 0.0;
        }
}
```

Java code of `LineItem` class is below. The `LineItem` class has `order` relationship property annotated with `ManyToOne`.

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

@Entity
public class LineItem   implements Serializable {

        private int id;
        private String product;
        private double amount;
        private double price;
        private Order order;
```

```
          @Id@ GeneratedValue(strategy=GenerationType.AUTO)
          public int getId() {
              return id;
          }
          public void setId(int id) {
              this.id = id;
          }
          public String getProduct() {
              return product;
          }
          public void setProduct(String product) {
              this.product = product;
          }
          public double getAmount() {
              return amount;
          }
          public void setAmount(double amount) {
              this.amount = amount;
          }
          public double getPrice() {
              return price;
          }
          public void setPrice(double price) {
              this.price = price;
          }
          @ManyToOne
          public Order getOrder() {
              return order;
          }
          public void setOrder(Order order) {
              this.order = order;
          }
          public LineItem(){}
          public LineItem(String product, double amount, double price){
             this.product = product;
             this.amount = amount;
             this.price = price;
          }
}
```

6.2.2 Unidirectional many-to- one relationships

Assuming that:

Entity A references a single instance of Entity B.
Entity B references a collection of Entity A.

Entity A must be the owner of the relationship.

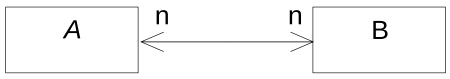
The following mapping defaults apply:

Entity A is mapped to a table named `A`.
Entity B is mapped to a table named `B`.
Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B`.

6.3 Many-to-many relationsip

Assuming that:

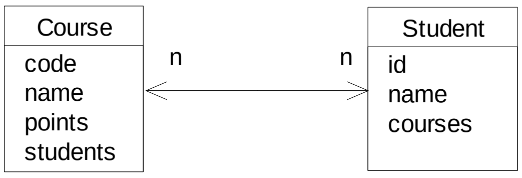
Entity A references a collection of Entity B.
Entity B references a collection of Entity A.
Entity A is the owner of the relationship.



The following mapping defaults apply:

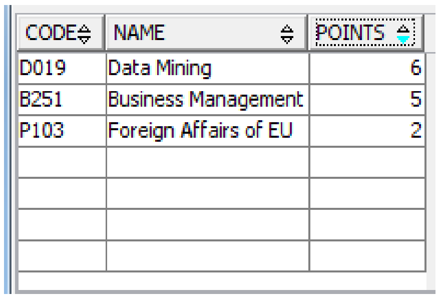
Entity A is mapped to a table named A.
Entity B is mapped to a table named B.
There is a join table that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity B; "_"; the name of the primary key column
in table A. The other foreign key column refers to table B and has the same type as the primary key of table B. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary
key column in table B.

Example
Course and Student entities have a bidirectional many-to-many relationship: a course can have many students and a student can participate in many courses. For the relationship the Course class has the students relationship property and the Student class has the courses relationship property. Because the relationship is many-to-many, both relationship properties are Collection type. Other properties of the entity classes are persistent properties. The Student entity is the owner of this relationship.



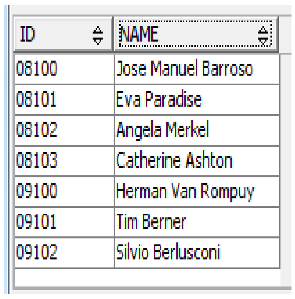
The entities with the relationship is mapped to the relation model in the following way. The Course entity is mapped to the COURSE table, the Student entity is mapped to the STUDENT table and the bidirectional many-to-many relationship is mapped to the STUDENT_COURSE join table (owner name first) with foreign key columns STUDENTS_ID and COURSES_CODE. (figure below)

COURSE table



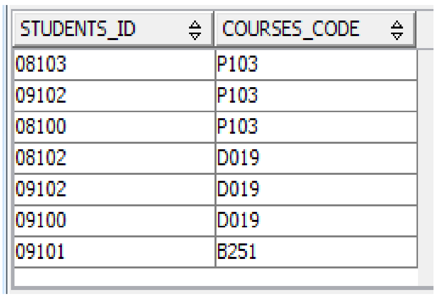
CODE	NAME	POINTS
D019	Data Mining	6
B251	Business Management	5
P103	Foreign Affairs of EU	2

STUDENT table



ID	NAME
08100	Jose Manuel Barroso
08101	Eva Paradise
08102	Angela Merkel
08103	Catherine Ashton
09100	Herman Van Rompuy
09101	Tim Berner
09102	Silvio Berlusconi

STUDENT_COURSE table



STUDENTS_ID	COURSES_CODE
08103	P103
09102	P103
08100	P103
08102	D019
09102	D019
09100	D019
09101	B251

The Java code of the `Course` class with `code`, `name` and `points` persistent properties and `students` relationship property annotated with the `ManyToMany` annotation and the `mappedBy` element is below.

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

@Entity
public class Course implements Serializable{
       private String code;
       private String name;
       private double points;
       private Collection<Student> students;

       @Id
       public String getCode() {
         return code;
       }
       public void setCode(String code) {
         this.code = code;
       }
       public String getName() {
         return name;
       }
       public void setName(String name) {
         this.name = name;
       }
       public double getPoints() {
         return points;
       }
       public void setPoints(double points) {
         this.points = points;
       }
       @ManyToMany(mappedBy="courses")
       public Collection<Student> getStudents() {
         return students;
       }
       public void setStudents(Collection<Student> students) {
         this.students = students;
       }
       public void addStudent(Student student){
         students.add(student);
       }
       public void dropStudent(Student student){
         students.remove(student);
       }
       public Course() {}
       public Course(String code, String name, double points) {
         this.code = code;
         this.name = name;
         this.points = points;
       }
}
```

The Java code of the Student class with id and name persistent properties and courses relationship property annotated with the ManyToMany annotation is below.

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

@Entity
public class Student implements Serializable {
    private String id;
    private String name;
    private Collection<Course> courses;

    @Id
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @ManyToMany
    public Collection<Course> getCourses() {
        return courses;
    }
    public void setCourses(Collection<Course> courses) {
        this.courses = courses;
    }
    public void addCourse(Course course){
        courses.add(course);
    }
    public void dropCourse(Course course){
        courses.remove(course);
    }
    public Student(){}
    public Student(String id, String name){
        this.id = id;
        this.name = name;
    }
}
```

7. Query Language

The Java Persistence query language is a string-based query language used to define queries over entities, their persistent state and relationships. Unlike SQL, this query language is truly platform-independent and object aware. This means that the entities and their fields are referenced by their names, rather than having to know the details of the table and column names in the underlying relational database.

The query language uses the abstract persistence schema of entities, including their embedded objects and relationships, for its data model, and it defines operators and expressions based on this data model. It uses a SQL-like syntax to select objects or values based on abstract schema types and relationships.

The syntax of the SELECT statement is following:

```
    SELECT clause1
    FROM clause2
    [WHERE clause3]
    [GROUP BY clause4]
    [HAVING clause5]
    [ORDER BY clause6]
```

The `SELECT` and `FROM` clauses are obligatory and the other clauses are optionally. The `SELECT` clause determines the type of the objects or values to be selected. The `FROM` clause provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply. The `WHERE` clause is used to restrict the results that are returned by the query. The `GROUP BY` clause allows query results to be aggregated in terms of groups. The `HAVING` clause allows filtering over aggregated groups. The `ORDER BY` clause is used to order the results that are returned by the query.

Example queries

Simple queries

The basic `SELECT` query is

```
SELECT [DISTINCT] p
FROM Product p
```

The result of the query is all products. The `SELECT` clause designates the return type of this query to be of type `Product`. The optional `DISTINCT` keyword eliminates duplicate values. The `FROM` clause declares an identification variable `p`. The above `FROM` clause is the abbreviation of `FROM Product AS p` clause. The identification variables can be declared only in the `FROM` clause, they can instead be used also in `SELECT` and `WHERE` clauses.

The following `SELECT` query has a `WHERE` clause

```
SELECT p
FROM Product p
WHERE p.productGroup = 'Wines'
```

The result of this query is a list of products, whose product group is Wines. The WHERE clause contains a path expression with the identification variable `p` followed by the navigation operation (`.`) and the persistent state field `productGroup` , whose value is the value of the `productGroup` field of the `Product` entity. This value is compared to the literal string 'Wines' and the value of the comparison (boolean value: `true` or `false`) determines, whether the entity is in the result of the query.

The following SELECT query contains the WHERE clause with comparison operators (>= and <) and logical operator (`AND`).

```
SELECT p
FROM Product p
WHERE p.price >= 0.0 AND p.price < 500.00
```

The result of the query is a list of products, whose price is equal or more than 0.0 and less than 500.00.

The query can have input parameters, which can only be used in the WHERE clause or HAVING clause of a query. A named parameter is denoted by an identifier that is prefixed by the ":" symbol. The `SELECT` query below has `:pgroup` input parameter.

```
SELECT p
FROM Product p
WHERE p.productGroup = :pgroup
```

The value of the input parameter is given by calling `setParameter()` method of `Query` object.

There is another type of input parameter, a positional parameter, which is designated by the question mark (`?`) prefix followed by an integer, for example: `?1`. The following query has two positional parameters.

```
SELECT p
FROM Product p
WHERE p.price >= ?1 AND p.price < ?2
```

Queries navigating to related entities

Two entities can be joined with `JOIN` keyword for navigating from an entity to another via a one-to-one relationship field. The following `SELECT` query navigates between `Order` and `Customer` entities via one-to-one `customer` relationship field of `Order` entity.

```
SELECT o
FROM Order o JOIN o.customer c
WHERE c.name = 'HAAGA-HELIA'
```

The result of the query is the orders of the customer, whose name is HAAGA-HELIA. The `FROM` clause is the abbreviation of `FROM Order AS o JOIN o.customer AS c`.

In one-to-many relationship, the multiple side consists of a collection of entities. A collection member declaration is declared by using `IN` operator. The following `SELECT` query navigates from the `Order` entity via its many-to-one relationship field `lineItems` to the `LineItem` entity and further to the `Product` entity.

```
SELECT o
FROM Order o, IN (o.lineItems) l JOIN l.product p
WHERE p.code = 'W0995'
```

The `SELECT` clause defines the types of the objects or values returned by the query. All the `SELECT` clauses of above queries have defined, that the type of the query result is the type of the entity. The `SELECT` query can return also persistent fields of the entity. The following `SELECT` query returns the names of the products ordered by Pedro Cavador.

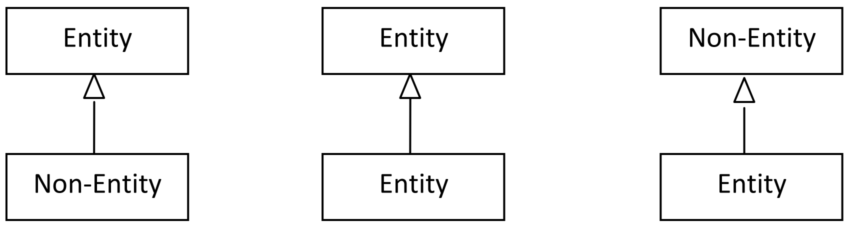
```
SELECT p.name
FROM Order o, IN (o.lineItems) l JOIN l.product p JOIN o.customer c
WHERE c.name = ‘Pedro Cavador’
```

This is a short introduction to the JPA query language, which is a comprehensive object based query language. More information can be found in the JPA 2.0 Specification and in other literature.

8. Inheritance of entity

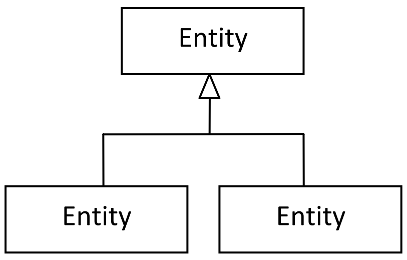
An entity may inherit from another entity class. Entities support inheritance, polymorphic associations, and polymorphic queries. Both abstract and concrete classes can be entities. Both abstract and concrete classes can be annotated with the Entity annotation and queried for as entities. An abstract entity class cannot have instancies.

Entities can extend non-entity classes and non-entity classes can extend entity classes.



Mapping of class hierarchies

How to map a class hierarchy to the relational database?



There are three basic strategies that are used when mapping a class or class hierarchy to a relational database:

- a single table per class hierarchy (default strategy)
- a joined subclass strategy, in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass.
- a table per concrete entity class

Single table per class hierarchy

All the classes in a hierarchy are mapped to a single table. The table has a column that serves as a “discriminator column”, that is, a column whose value identifies the specific subclass to which the instance that is represented by the row belongs.

This mapping strategy provides good support for polymorphic relationships between entities and for queries that range over the class hierarchy.

Joined subclass

In the joined subclass strategy, the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains those fields that are specific to the subclass (not inherited from its superclass).

This strategy provides support for polymorphic relationships between entities.

Table per concrete entity class

Each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

This strategy provides poor support for polymorphic relationships.

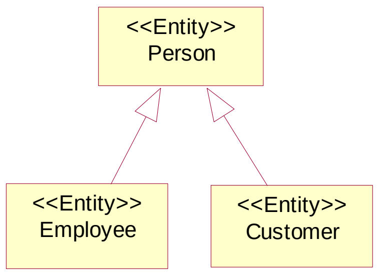
Annotation of mapping class hierarchy

The mapping of class hierarchies is specified through metadata. `Inheritance` annotation defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy. Inheritance annotation has a `strategy` element, which is used for setting the entity inheritance hierarchy strategy. For the inheritance strategy options JPA has an `InheritanceType` class with constant values:

```
InheritanceType.SINGLE_TABLE
InheritanceType.JOINED
 InheritanceType.TABLE_PER_CLASS.
```

`SINGLE_TABLE` is default setting of inheritance stategy.

Example.
The class model consists of a super class `Person` and its two subclasses `Employee` and `Customer`.



The entities in Java are below:
Person:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Person implements Serializable {
        private String code;
        private String name;
        private String address;
        private String degree;
        private String gender;

        @Id
        public String getCode() {
             return code;
        }
        public void setCode(String code) {
             this.code = code;
        }
        public String getName() {
             return name;
        }
        public void setName(String name) {
             this.name = name;
        }
        public String getAddress() {
             return address;
        }
        public void setAddress(String address) {
             this.address = address;
        }
        public String getGender() {
             return gender;
        }
        public void setGender(String gender) {
             this.gender = gender;
        }
        public String getDegree() {
             return degree;
        }
        public void setDegree(String degree) {
             this.degree = degree;
        }
        public Person() {}
}
```

Employee:

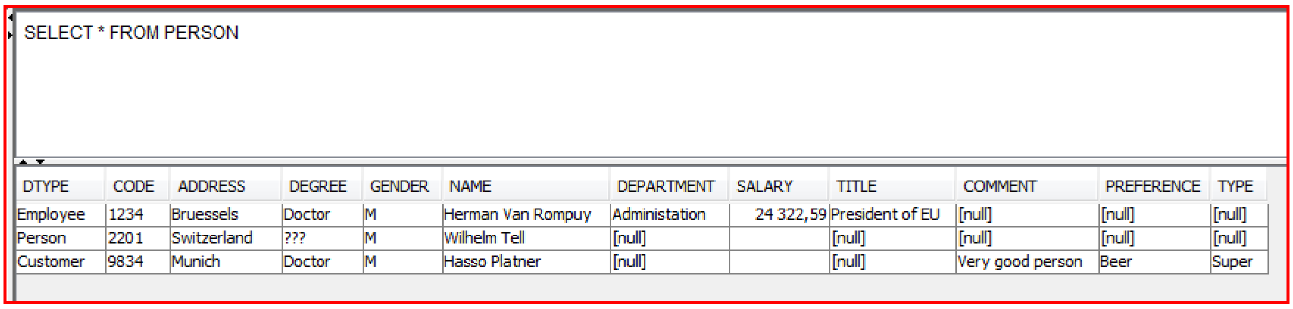
```
@Entity
public class Employee extends Person{
        private String title;
        private String department;
        private double salary;

        public String getTitle() {
              return title;
        }
        public void setTitle(String title) {
              this.title = title;
        }
        public String getDepartment() {
              return department;
        }
        public void setDepartment(String department) {
              this.department = department;
        }
        public double getSalary() {
              return salary;
        }
        public void setSalary(double salary) {
              this.salary = salary;
        }
}
```

Customer:

```
@Entity
public class Customer extends Person {
        private String type;
        private String preference;
        private String comment;

        public String getType() {
              return type;
        }
        public void setType(String type) {
              this.type = type;
        }
        public String getPreference() {
              return preference;
        }
        public void setPreference(String preference) {
              this.preference = preference;
        }
        public String getComment() {
              return comment;
        }
        public void setComment(String comment) {
              this.comment = comment;
        }
}
```

Inheritance strategy `SINGLE_TABLE` of JPA maps all three classes to a relational table with the default name `PERSON`, which is the name of root class of the class hierarchy.

The name of the first column of the `PERSON` table is `DTYPE`, which describes the class of the entity instance. Every fields of every classes of the whole class hierarchy is mapped to the columns of a table. The unused columns of the rows in the table has value `null`.



```sql
SELECT * FROM PERSON
```

DTYPE	CODE	ADDRESS	DEGREE	GENDER	NAME	DEPARTMENT	SALARY	TITLE	COMMENT	PREFERENCE	TYPE
Employee	1234	Bruessels	Doctor	M	Herman Van Rompuy	Administation	24 322,59	President of EU	[null]	[null]	[null]
Person	2201	Switzerland	???	M	Wilhelm Tell	[null]		[null]	[null]	[null]	[null]
Customer	9834	Munich	Doctor	M	Hasso Platner	[null]		[null]	Very good person	Beer	Super

Referencies

Bauer, C. & King, G. 2007. Java Persistence with Hibernate. Manning Publication Co.

King G., Bauer C., Andersen M. R., Bernard, E. & Ebersole S. 2009. Hibernate Reference Documentation. Red Hat Middleware. (http://docs.jboss.org/hibernate/core/3.3/reference/en/html/)

Sriganesh, R., P. 2006. Mastering Enterprise JavaBeans 3.o. Wiley Publishing, Inc.

Sun Microsystems. 2009. JSR 317: Java Persistence API, Version 2.0. Final Release.