Java Persistence API

Jeszenszky, Péter
University of Debrecen, Faculty of Informatics
jeszenszky.peter@inf.unideb.hu

Kocsis, Gergely
University of Debrecen, Faculty of Informatics
kocsis.gergely@inf.unideb.hu

Last modified: 26.04.2017

Definitions

- *Persistence*
- *Data access object* – DAO
- *Domain model*
- *Anemic domain model*
- *Plain old Java object* – POJO
- JavaBean
- *object-relational mapping* – ORM

Persistence

- Meaning: the state of occurring or existing beyond the usual, expected, or normal time (http://www.merriam-webster.com/dictionary/persistence)
- In computer science it is used to data that overlives the process that created it.

How to implement persistence

- Java provides several different ways to implement persistence:
  - File management
  - Java Architecture for XML Binding (JAXB)
  - JDBC
  - Object serialization: see `java.io.Serializable` interface http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html
  - …
- Hereafter we focus on storing data in relational databases

Data access object (DAO)

- It defines an interface which through persistence operations can be executed on a given entity
- See:
  - Deepak Alur, Dan Malks, John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. 2nd edition. Prentice Hall, 2003.
  - *Core J2EE Patterns – Data Access Object* http://corej2eepatterns.com/DataAccessObject.htm

Domain model

- Object model of a specific domain. It contains data and behavior as well. http://martinfowler.com/eaaCatalog/domainModel.html
  - See:
    - Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

Anemic domain model

- A domain model that does not contain behavior
  - Sometimes it is treated as an antipattern

Martin Fowler

- See:
  - Martin Fowler: *AnemicDomainModel*. http://www.martinfowler.com/bliki/AnemicDomainModel.html

POJO – Plain Old Java Object

- A concept by Rebecca Parsons, Josh MacKenzie and Martin Fowler (2000)
- It is a simple Java object for which no restrictions are given
  - E.g. There no mandatory interfaces to implement
- See:
  - Martin Fowler: *POJO*. http://www.martinfowler.com/bliki/POJO.html

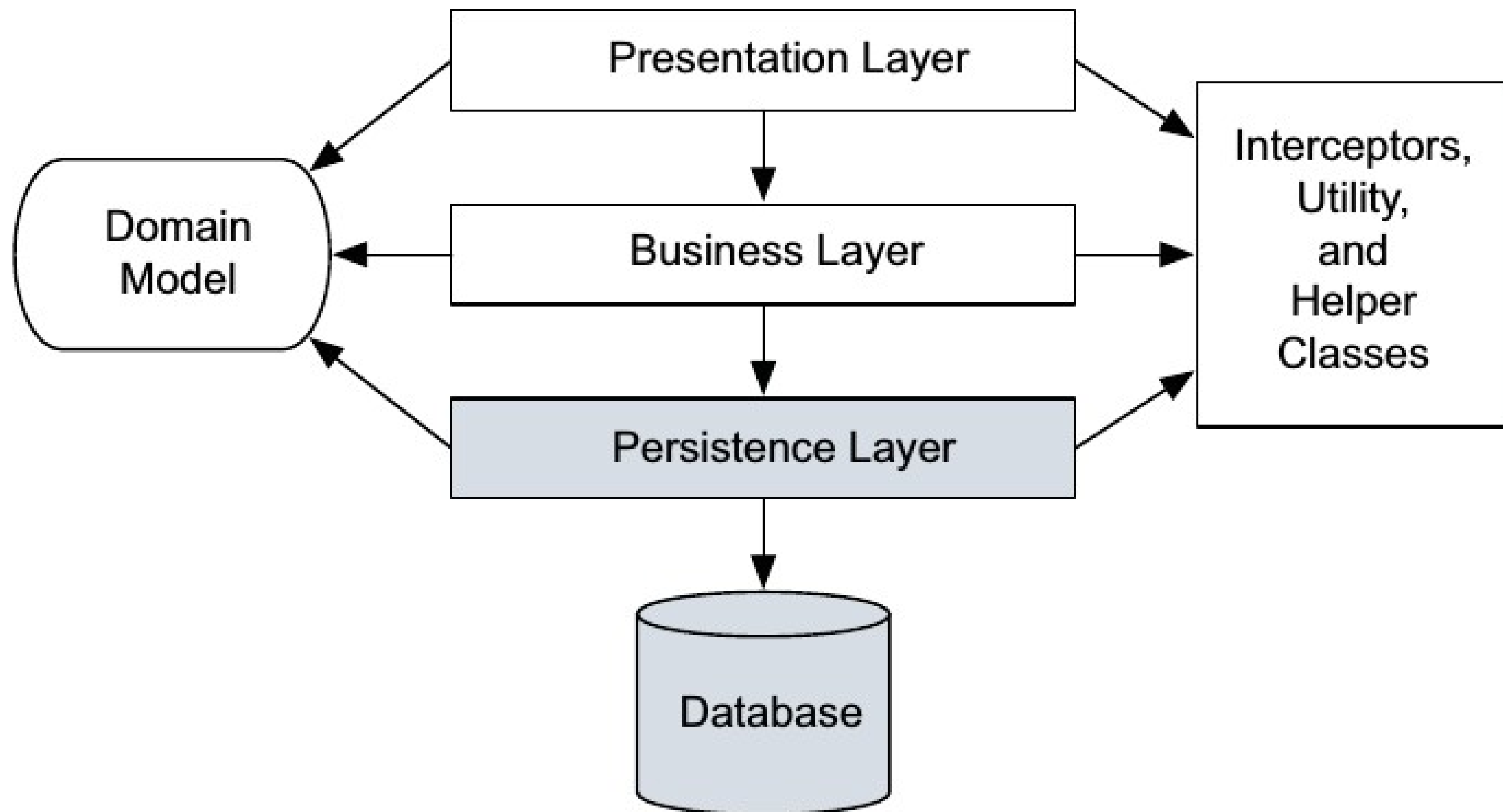
JavaBean

- A class implementing the `java.io.Serializable` interface. It has to contain a default constructor (parameterless) and has to provide getters and setters for its fields
  - See:
    - *JavaBeans Specification 1.01 (Final Release)* (August 8, 1997) http://www.oracle.com/technetwork/articles/javaee/spec-136004.html
    - *The Java Tutorials – Trail: JavaBeans* http://docs.oracle.com/javase/tutorial/javabeans/
    - Stephen Colebourne. *The JavaBeans specification*. November 28, 2014. http://blog.joda.org/2014/11/the-javabeans-specification.html

Object-relational mapping – ORM

- It provides conversion between object oriented programming languages and relational database tables
  - An implementation of storing objects of domain models in relational database tables

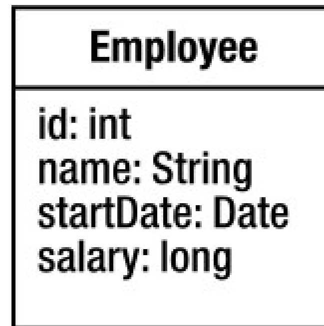
Layered application architecture

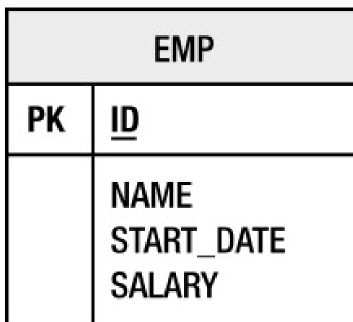


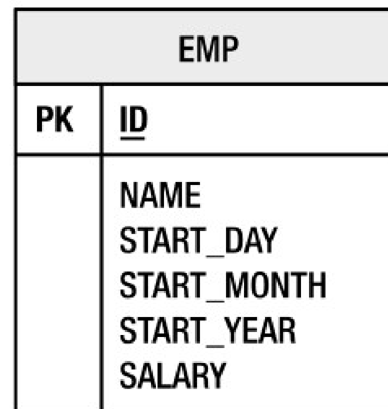
Paradigm collision

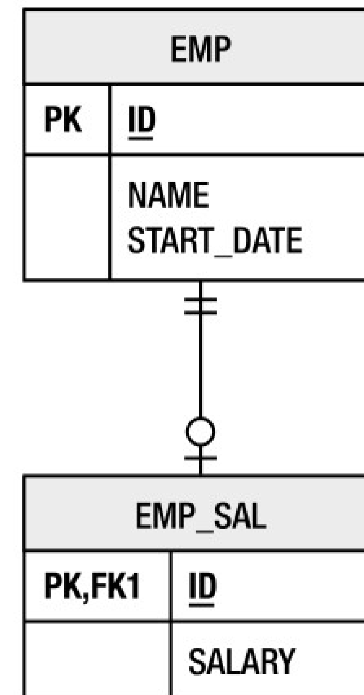
- *Object-relational impedance mismatch*:
  - It means the difficulties arising as a result of the differences between the two different world
  - There are different concepts and sometimes there is no suitable pair of one in the other world
    - Example:
      - Associations: In order to represent N:M connections a connecting table is required. However in the object model this concept is not present

Implementation of Object-relational mapping (1)

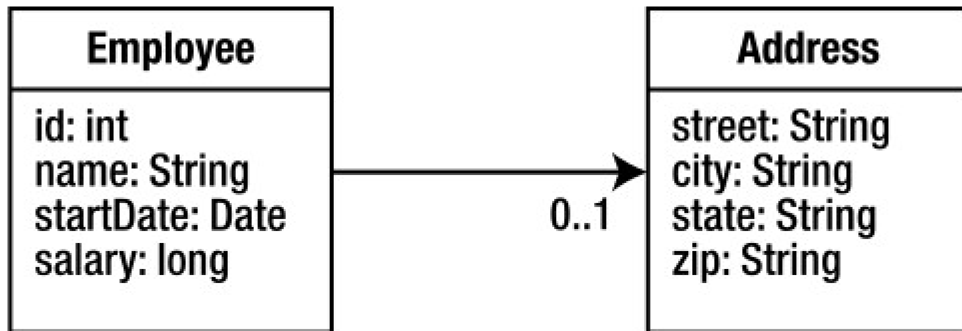




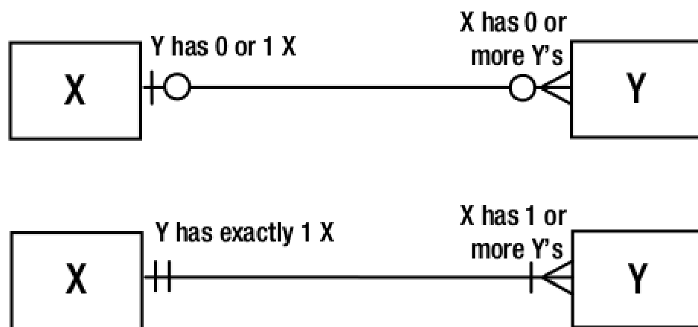




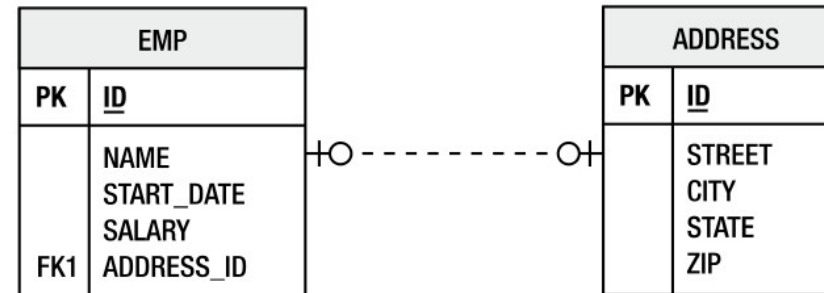
Implementation of Object-relational mapping (2)



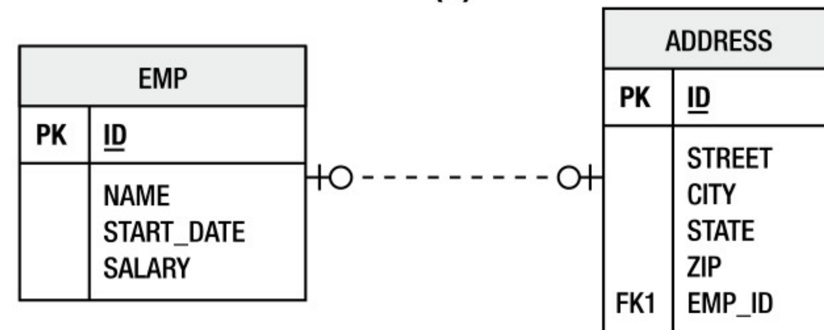
Legend:



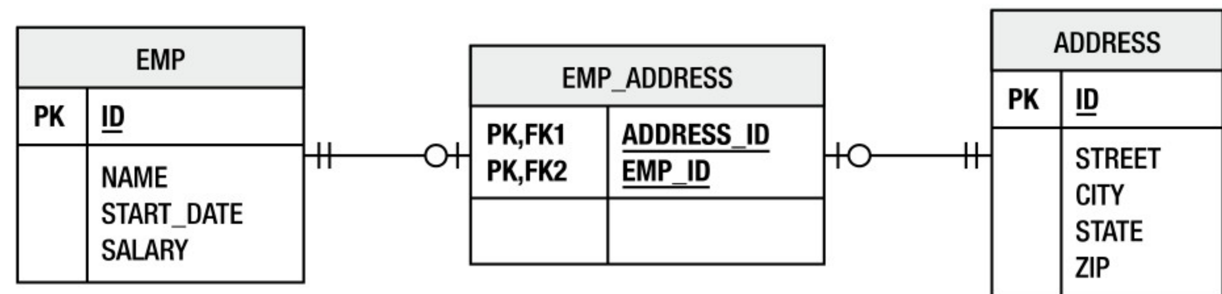
(A)



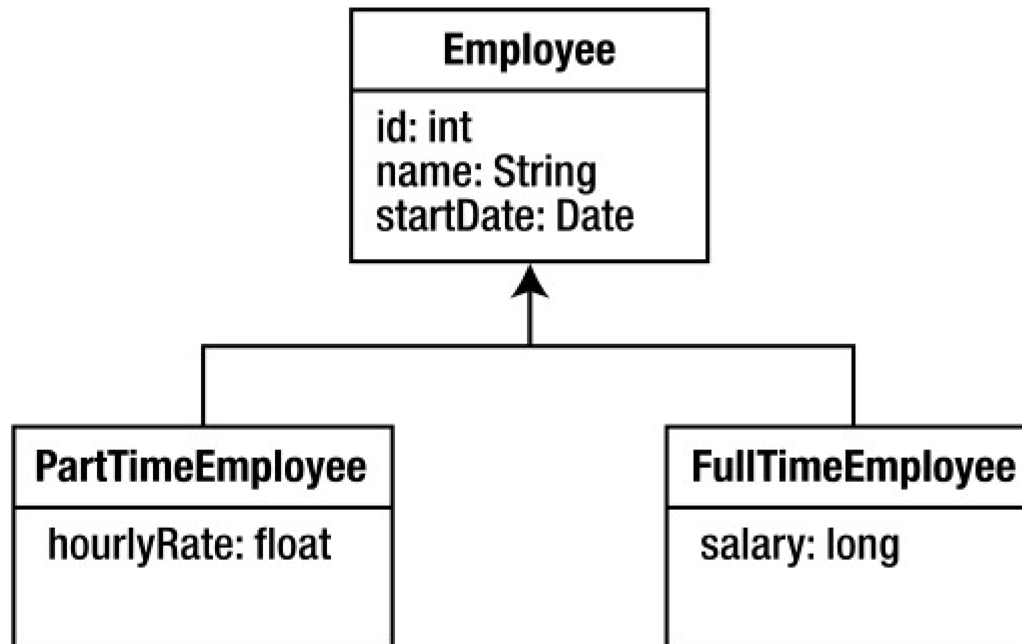
(B)



(C)

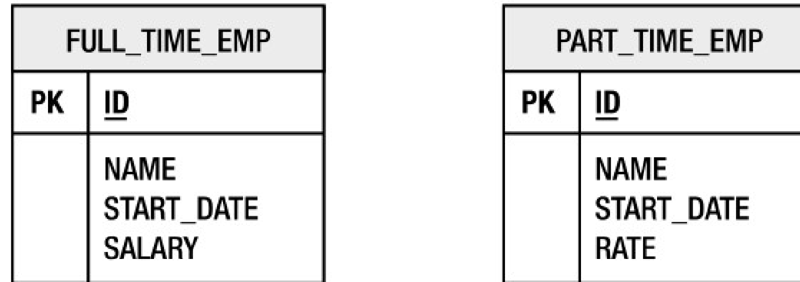


Implementation of Object-relational mapping (3)

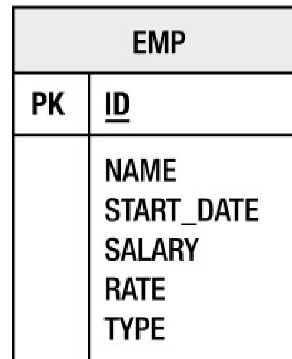


Implementation of Object-relational mapping (4)

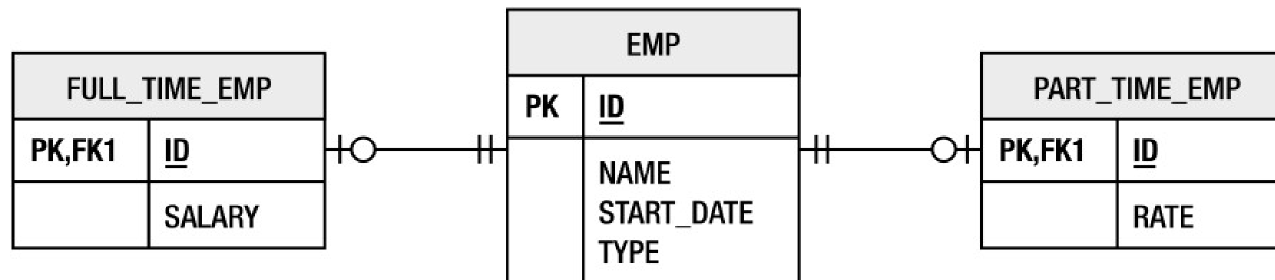
(A)



(B)



(C)



Earlier implementations of object persistence in Java (1)

- **Producer dependent solutions**:
  - Open source Free solutions (e.g. Castor, Hibernate)
  - Commercial solutions (TopLink)
- **JDBC**
- ***Data mappers***:
  - Partial solutions half way between JDBC and full ORM solutions. The application developer provides the SQL statements
    - Example: MyBatis (license: Apache Software License) http://www.mybatis.org/mybatis-3/
  - See:
    - Martin Fowler: *Data Mapper*. http://martinfowler.com/eaaCatalog/dataMapper.html
    - Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

Earlier implementations of object persistence in Java (2)

- ***Enterprise JavaBeans – Entity Beans****:*
  - Since the introduction of EJB 3.0 in 2006 it is deprecated and obsoleted by JPA
- ***Java Data Objects*** (***JDO***):
  - Example:
    - Apache JDO (license: Apache License v2) https://db.apache.org/jdo/
    - DataNucleus Access Platform (licenc: Apache License v2) http://www.datanucleus.org/

Java Persistence API (1)

- POJO-based ORM solution of Java object persistence
  - Originally introduced in Enterprise JavaBeans (EJB) 3.0 specification in 2006 as a part of Java EE 5
  - Current version: 2.1 introduced in 2013

Java Persistence API (2)

- Contained by the following packages:
  - `javax.persistence` https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html
  - `javax.persistence.criteria` https://docs.oracle.com/javaee/7/api/javax/persistence/criteria/package-summary.html
  - `javax.persistence.metamodel` https://docs.oracle.com/javaee/7/api/javax/persistence/metamodel/package-summary.html
  - `javax.persistence.spi` https://docs.oracle.com/javaee/7/api/javax/persistence/spi/package-summary.html

Specification

- *JSR 338: Java Persistence 2.1 (Final Release)* (22 May, 2013) https://jcp.org/en/jsr/detail?id=338

Properties (1)

- **POJO-based**
- ***Non-intrusiveness***
  - The persistence API is separated from the persistent classes
    - The business logic of the application calls it. It passes the persistent objects as parameters to the API.
- **Object oriented query language**
  - Java Persistence Query Language (JPQL)

Properties (2)

- **Mobil entities**
  - The persistent objects can be transmitted from one Java virtual machine to another
- **Simple configuration**
  - There are default settings for everything. Only the exceptional cases needed to be configured (*configuration by exception*)
  - For the mapping meta-data can be provided by annotations or in a separate XML document
- **No application server is needed**
  - Can be used in Java SE as well (not only in Java EE)

JPA implementations

- Open Source Free software:
  - DataNucleus (licenc: Apache License v2) http://www.datanucleus.org/
  - EclipseLink (license: Eclipse Public License) http://www.eclipse.org/eclipselink/
    - Reference implementation of JPA 2.1
    - GlassFish Server uses it a default persistence solution
  - Hibernate (licenc: GNU LGPL v2.1) http://hibernate.org/
    - WildFly uses it a default persistence solution https://docs.jboss.org/author/display/WFLY10/JPA+Reference+Guide

IDE support

- Eclipse:
  - Dali Java Persistence Tools (license: Eclipse Public License) https://eclipse.org/webtools/dali/
    - A part of Eclipse IDE for Java EE Developers
- NetBeans:
  - NetBeans JPA Modeler (license: Apache License v2) http://jpamodeler.github.io/ http://plugins.netbeans.org/plugin/53057/jpa-modeler

Hibernate components

- Hibernate OGM http://hibernate.org/ogm/
  - JPA support for NoSQL databases (e.g. MongoDB, Neo4j)
- Hibernate ORM http://hibernate.org/orm/
  - JPA implementation
- Hibernate Search http://hibernate.org/search/
  - Full text search support
- Hibernate Validator http://hibernate.org/validator/
  - Annotation-based solution to validate restrictions on objects
- Hibernate Tools http://hibernate.org/tools/
  - For developers (Eclipse extensions and Apache Ant task)
- …

Supported Database management systems

- See `org.hibernate.dialect` package http://docs.jboss.org/hibernate/orm/5.1/javadocs/org/hibernate/dialect/package-summary.html
  - Microsoft SQL Server, Oracle, MySQL, PostgreSQL, Apache Derby (Java DB), H2, HSQLDB, …

Availability

- Available in Maven central repository (last stable version `5.1.0.Final`)
  - Products:
    - `org.hibernate:hibernate-core:5.1.0.Final`
    - `org.hibernate:hibernate-java8:5.1.0.Final`
    - `org.hibernate:hibernate-tools:5.1.0.Alpha3`
    - `org.hibernate:hibernate-gradle-plugin:5.1.0.Final`
    - ...

Maven support

- Hibernate Maven Plugin
  http://juplo.de/hibernate-maven-plugin/
  - Available in Maven central repository:
    `de.juplo:hibernate-maven-plugin:2.0.0`

NHibernate

- NHibernate (programming language: C#, license: GNU GPL v2.1) http://nhibernate.info/
  - A Hibernate port for .NET platform

Entity

- A lightweight persistent domain specific object

Entity class

- A class marked by `javax.persistence.Entity` annotation or a class marked as an entity class in the XML descriptor file
- It has a mandatory `public` or `protected` default constructor
- It has to be a top level class. Must not be `enum` or interface
- For the class, its methods and persistent instance variables `final` modifier cannot be used

Persistent fields and attributes (1)

- The persistent state of an object is represented by the instance variables that fulfill JavaBean properties.
- Instance variables can be reached only through the methods by the clients of the entity
- The instance variables of the entity can be reached by the persistence provider running environment through the `getters` and `setters`
  - The visibility of instance variables can be `private`, `protected` or package private (no modifier)
  - If the instance variables can be reached only through the getters and setters, the visibility of these have to be `public` or `protected`

Persistent fields and attributes (2)

- Collection type persistent fields and attributes have to be of type `java.util.List`, `java.util.Map` or `java.util.Set`

Primary keys and entity identity

- All entities must have a primary key
- The primary key has to be one or more instance variables or attributes of the entity class
  - The primary key can be simple or compound (composite)
- The type of the instance variables and attributes forming the primary key can be:
  - Primitive types and their respective Object types, `java.lang.String`, `java.util.Date`, `java.sql.Date`, `java.math.BigDecimal`, `java.math.BigInteger`

EntityManager (1)

- The `javax.persistence.EntityManager` interface provides an API for persistence operations https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html
  - Example `find()`, `persist()` and `remove()` methods

EntityManager (2)

- When an `EntityManager` gets a reference to an entity object we say that the `EntityManager` **manages** the object
  - The reference to the object can be given as a result of a method call or by reading from a database
- The objects manages by the `EntityManager` are called the **Persistence context**
- An `EntityManager` manages specific type entities stored in a specific database
  - The types of the managed entities are defined by a **persistence unit**

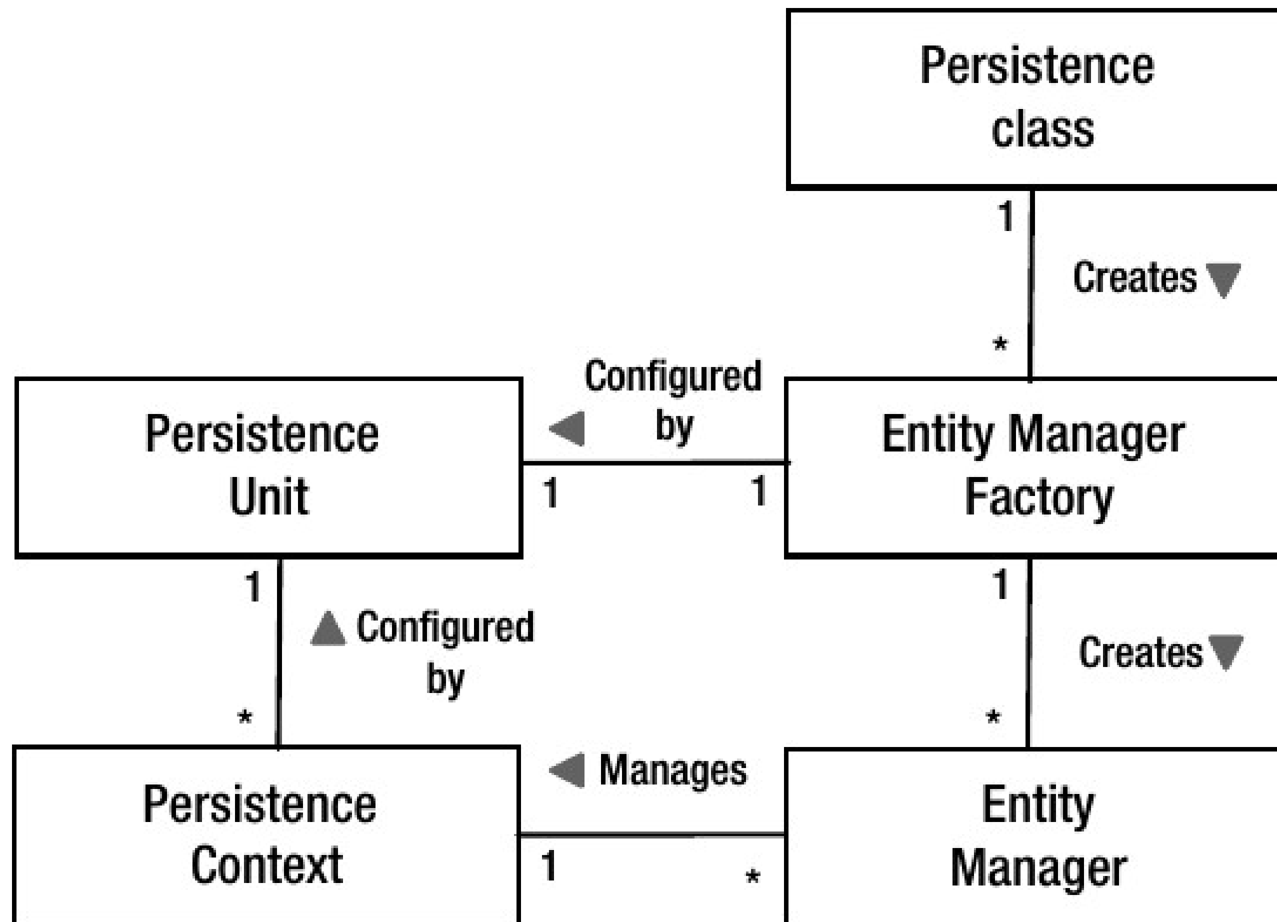
Persistence unit

- A Persistence Unit is a logical grouping of persistable classes along with their related settings that are mapped to the same database.

EntityManagerFactory

- The `javax.persistence.EntityManagerFactory` interface can provide `EntityManager` objects for a given persistence unit https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManagerFactory.html
  - In Java SE applications an instance of the `EntityManagerFactory` object can be created by the use of the `createEntityManagerFactory()` method of the `javax.persistence.Persistence` class https://docs.oracle.com/javaee/7/api/javax/persistence/Persistence.html

JPA concepts



Persistence unit (more) (1)

- A logical group containing the followings:
  - An `EntityManagerFactory` and its `EntityManager`-s together with the configuration settings
  - The set of classes managed by the persistence unit (by the `EntityManager-s of the EntityManagerFactory EntityManager`)
  - ORM metadata (annotations and/or XML metadata form)
- Its definition is done in the `persistence.xml` file
- Every persistence unit has a name
  - In the scope of one aplication it is advised to give unique names for all persistence units

persistence.xml (1)

- An XML document defining one or more persistence units
  - XML scheme: http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd
- The root of the persistence unit has to be in the `META-INF` folder

persistence.xml (2)

- File structure:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
        version="2.1">
    <!-- One or more persistence-unit element: -->
    <persistence-unit name="név">
        <!-- ... -->
    </persistence-unit>
    <!-- ... -->
</persistence>
```

persistence.xml (3):

- Example (1): an EAR (Enterprise Archive) file

```
emp.ear:
    emp-ejb.jar:
        META-INF/persistence.xml
        pkg/ejb/EmployeeService.class
    lib/emp-classes.jar:
        META-INF/orm.xml
        META-INF/emp-mappings.xml
        pkg/model/Employee.class
        pkg/model/Phone.class
        pkg/model/Address.class
        pkg/model/Department.class
        pkg/model/Project.class
```

persistence.xml (4)

- Example (2): a `META-INF/persistence.xml` file

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        version="2.1">
    <persistence-unit name="EmployeeService">
        <jta-data-source>java:app/jdbc/EmployeeDS</jta-data-source>
        <mapping-file>META-INF/emp-mappings.xml</mapping-file>
        <jar-file>lib/emp-classes.jar</jar-file>
    </persistence-unit>
</persistence>
```

XML ORM descriptor (1)

- An XML document holding the metadata for the object-relational mapping. By the use of it the metadata given by annotations can be overwritten
  - If the `persistence-unit-metadata/xml-mapping-metadata-complete` element is present annotations are to be omitted
    - In other cases the document overwrites or complete the metadata given by annotations
- XML scheme: http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd

XML ORM descriptor (2)

- File structure:

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
        http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
        version="2.1">
    <!-- ... -->
</entity-mappings>
```

XML ORM descriptor (3)

- Példa: a `META-INF/emp-mappings.xml` állomány:

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        version="2.1">
    <package>pkg.model</package>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <schema>HR</schema>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <entity class="Employee">
        <table name="EMP"/>
    </entity>
    <!-- ... -->
</entity-mappings>
```

JPA „Hello, world!” program (1)

- `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.1.0.Final</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.1.0.Final</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derby</artifactId>
        <version>10.12.1.1</version>
        <scope>runtime</scope>
    </dependency>
```

JPA „Hello, world!” program (2)

- `Message.java`:

```
package hello.model;

@javax.persistence.Entity
public class Message {

    @javax.persistence.Id
    @javax.persistence.GeneratedValue
    private Long id;
    private String text;

    public Message() {
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

JPA „Hello, world!” program (3)

- `HelloJPA.java`:

```
package hello;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import hello.model.Message;

public class HelloJPA {

    public static void main(String[] args) {
        EntityManagerFactory emf =
                    Persistence.createEntityManagerFactory("HelloPU");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Message message = new Message();
        message.setText("Hello, world!");
        em.persist(message);
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

JPA „Hello, world!” program (4)

- `META-INF/persistence.xml`:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
version="2.1">
    <persistence-unit name="HelloPU">
        <class>hello.model.Message</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:testDB;create=true"/>
            <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
        </properties>
    </persistence-unit>
</persistence>
```

JPA „Hello, world!” program (5)

- Modification in the database:

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("HelloPU");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
List<Message> messages = em.createQuery("select m from
                hello.model.Message m").getResultList();
assert messages.size() == 1;
assert messages.get(0).getText().equals("Hello, world!");
messages.get(0).setText("Hello, JPA!");
em.getTransaction().commit();
em.close();
emf.close();
```

The same example without annotations (1)

- `Message.java`:

```
package hello.model;

public class Message {

    private Long id;
    private String text;

    public Message() {
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

The same example without annotations (2)

- META-INF/persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
version="2.1">
    <persistence-unit name="HelloPU">
        <mapping-file>META-INF/orm.xml</mapping-file>
        <class>hello.model.Message</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:testDB;create=true"/>
            <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
        </properties>
    </persistence-unit>
</persistence>
```

The same example without annotations (3)

- `META-INF/orm.xml`:

```
<entity-mappings
xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm",ver
sion="2.1">
    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
    </persistence-unit-metadata>
    <entity class="hello.model.Message" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
            <basic name="name"/>
        </attributes>
    </entity>
</entity-mappings>
```

A more complex example (1)

- `Employee.java`:

```
@@javax.persistence.Entity
public class Employee {

    @@javax.persistence.Id
    private int id;
    private String name;
    private long salary;

    public Employee() {}

    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 long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary = salary; }

}
```

A more complex example (2)

- `EmployeeService.java`:

```
import javax.persistence.*;
import java.util.List;

public class EmployeeService {

    protected EntityManager em;

    public EmployeeService(EntityManager em) {
        this.em = em;
    }

    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee();
        emp.setId(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }
```

A more complex example (3)

- `EmployeeService.java` (part 2):

```
public void removeEmployee(int id) {
    Employee emp = findEmployee(id);
    if (emp != null) {
        em.remove(emp);
    }
}

public Employee raiseEmployeeSalary(int id, long raise) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        emp.setSalary(emp.getSalary() + raise);
    }
    return emp;
}
```

A more complex example (4)

- `EmployeeService.java` (part 3):

```
    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }

    public List<Employee> findAllEmployees() {
        TypedQuery<Employee> query = em.createQuery(
                "SELECT e FROM Employee e", Employee.class);
        return query.getResultList();
    }
}
```

A more complex example (5)

- `EmployeeTest.java`:

```
import javax.persistence.*;
import java.util.List;

public class EmployeeTest {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        EmployeeService service = new EmployeeService(em);

        // create or save an employee
        em.getTransaction().begin();
        Employee emp = service.createEmployee(158, "John Doe", 45000);
        em.getTransaction().commit();
        System.out.println("Persisted " + emp);
```

A more complex example (6)

- `EmployeeTest.java` (part 2):

```
// load an employee
emp = service.findEmployee(158);
System.out.println("Found " + emp);

// load all employees
List<Employee> emps = service.findAllEmployees();
for (Employee e : emps) {
    System.out.println("Found employee: " + e);
}
```

A more complex example (7)

- `EmployeeTest.java` (part 3):

```
        // modify employee
        em.getTransaction().begin();
        emp = service.raiseEmployeeSalary(158, 1000);
        em.getTransaction().commit();
        System.out.println("Updated " + emp);

        // delete employee
        em.getTransaction().begin();
        service.removeEmployee(158);
        em.getTransaction().commit();
        System.out.println("Removed Employee 158");

        em.close();
        emf.close();
    }
}
```

Java 8 date and time types (1)

- The default support of Java 8 date and time was introduced in ORM 5.2.0
- For earlier versions `hibernate-java8` program library has to be used in runtime
  - See: *Hibernate ORM 5.2 release*. Jun 1, 2016. http://in.relation.to/2016/06/01/hibernate-orm-520-final-release/

Java 8 date and time types (2)

- `pom.xml`:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-java8</artifactId>
        <version>5.1.0.Final</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Java 8 date and time types (2)

- `Employee.java`:

```
@javax.persistence.Entity
public class Employee {

    @javax.persistence.Id
    private int id;
    private String name;
    private long salary;
    private java.time.LocalDate dob;

    public Employee() {}

    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 long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary = salary; }

    public LocalDate getDob() { return dob; }
    public void setDob(LocalDate dob) { this.dob = dob; }
}
```

Change table name

- `Employee.java`:

```
@javax.prsistence.Entity
@javax.pesistence.Table(name="EMP",
schema="HR")
public class Employee { ... }
```

Change default column name

- `Employee.xml`:

```
@javax.persistence.Entity
public class Employee {

    @javax.persistence.Id
    @javax.persistence.Column(name="EMP_ID")
    private int id;

    private String name;

    @javax.persistence.Column(name="SAL")
    private long salary;

    @javax.persistence.Column(name="COMM")
    private String comments;
    // ...
}
```

Lazy loading

- `Employee.xml`:

```
@javax.persistence.Entity
public class Employee {

    // ...
    @javax.persistence.Basic(fetch=FetchType.LAZY)
    @javax.persistence.Column(name="COMM")
    private String comments;
    // ...
}
```

The use of enums

```java
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,
    PART_TIME_EMPLOYEE,
    CONTRACT_EMPLOYEE
}

@Entity
public class Employee {

    @Id private long id;

    private EmployeeType
type;
    // ...
}
```

```java
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,
    PART_TIME_EMPLOYEE,
    CONTRACT_EMPLOYEE
}

@Entity
public class Employee {

    @Id private long id;

    @Enumerated(EnumType.STRING)
    private EmployeeType type;
    // ...
}
```

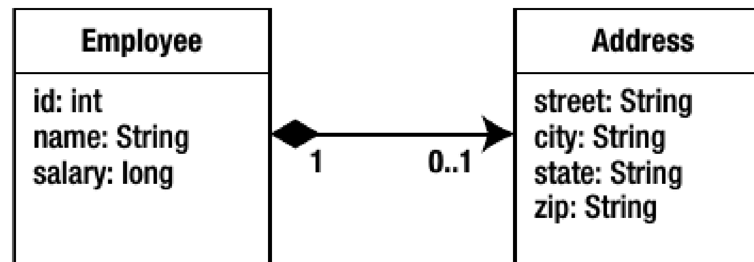
Embedded objects (1)

- Objects that do not have identity, but they are a part of another entity

Embedded objects (2)

- Example:

EMPLOYEE	
PK	ID
	NAME SALARY STREET CITY STATE ZIP_CODE



Embedded objects (3)

- Example:

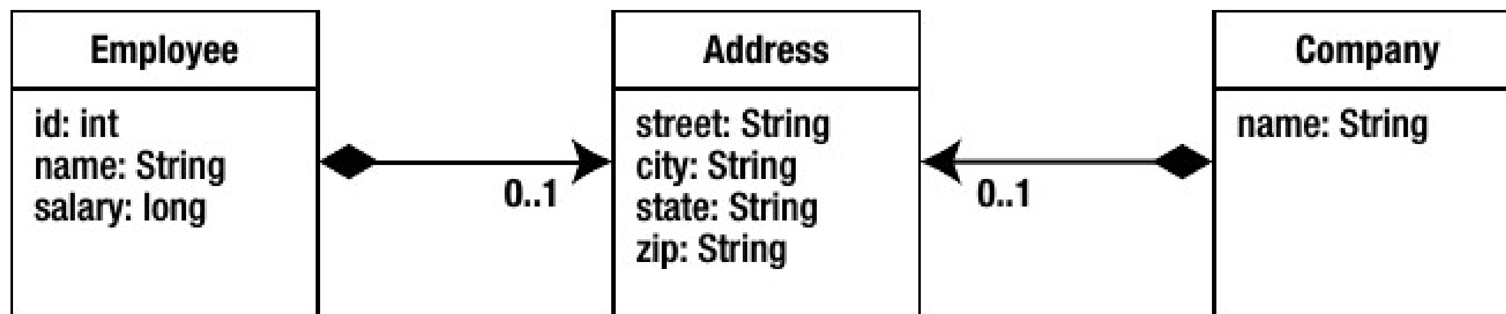
```
@Embeddable
@Access(AccessType.FIELD)
public class Address {

    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE") private String zip;
    // ...
}

@Entity
public class Employee {

    @Id private long id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}
```

Embedded objects (4)

- Example:

EMPLOYEE	
PK	ID
	NAME SALARY STREET CITY PROVINCE POSTAL_CODE

COMPANY	
PK	NAME
	STREET CITY STATE ZIP_CODE



Embedded objects (5)

- Example:

```
@Entity
public class Employee {

    @Id private long id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state",column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
    })
    private Address address;
    // ...
}

@Entity
public class Company {

    @Id private String name;
    @Embedded private Address address;
    // ...
}
```

Constraints (1)

- The restrictions of the `javax.validation.constraints` package can be used:
  - Bean Validation http://beanvalidation.org/
    - a part of Java EE (see javax.validation package and sub-packages)
      - *JSR 349: Bean Validation 1.1 (Final Release)* (24 May 2013) https://jcp.org/en/jsr/detail?id=349
    - Reference implementation: Hibernate Validator http://hibernate.org/validator/

Constraints (2)

- `pom.xml`:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>5.2.3.Final</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>javax.el</groupId>
        <artifactId>javax.el-api</artifactId>
        <version>2.2.5</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>javax.el</artifactId>
        <version>2.2.5</version>
        <scope>runtime</scope>
    </dependency>
    ...
</dependencies>
```

Constraints (3)

- `Employee.java`:

```
import javax.validation.constraints.*;

@javax.persistence.Entity;
public class Employee {

    @id
    private int id;

    @NotNull(message="Employee name must be specified")
    @Size(max=100)
    private String name;

    @NotNull
    @Size(max=40)
    @org.hibernate.validator.constraints.Email
    private String email;
    // ...
}
```

Constraints (4)

- Validation can be done:
  - Automatically before the execution of DB operations
    - By default automatic validation is done before `INSERT` and `UPDATE` operations
  - By calling the `validate()` method of a `javax.validation.Validator` object `validate()`
    - The object on which the validation is to be done has to be passed as a parameter

Constraints (5)

- Enabling automatic validation in `persistence.xml` file:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        version="2.1">
    <persistence-unit name="...">
        <validation-mode>AUTO|CALLBACK|NONE</validation-mode>
        <!-- ... -->
    </persistence-unit>
</persistence>
```

Constraints (6)

- Possible values of `validation-mode` :
  - **`AUTO`** (default): If a Bean Validation server is available automatic validation is done. Else no validation
  - **`CALLBACK`**: automatic validation is enabled, but if there is no available Bean Validation server an Exception is thrown
  - **`NONE`**: Disable automatic validation

Constraints (7)

- During validation the violation of constraints result in throwing `javax.validation.ConstraintViolationException` exception
- Hibernate uses the constraints to create the database schema

EntityManager operations (1)

- `void persist(Object entity)`:
  - Save the object given as a parameter to the database
    - If the object is already managed by the `EntityManager` no database operations are executed
    - Persisting an already present object in the database results in `javax.persistence.EntityExistsException`

EntityManager operations (2)

- `<T> T find(Class<T> entityClass, Object primaryKey)`:
  - Load an entity with a given primary key from the database
    - If the entity does not exist it returns `null`

EntityManager operations (3)

- `void remove(Object entity)`:
  - Remove the entity given as a parameter from the database
    - Only entities managed by the `EntityManager` can be removed, otherwise `IllegalArgumentException` is thrown

EntityManager operations (4)

- `void detach(Object entity)`:
  - Delete the entity given as a parameter from the persistence context
    - After it the object is called to be **detached**
    - The modifications not saved to the database are not synchronized any more

EntityManager operations (5)

- `<T> T merge(T entity)`:
  - Merge the (detached) entity to the persistence context. The opposite of detaching.
    - The entity given as a parameter stays detached but...
    - The returned value is a managed instance that has they same state as the parameter
      - This is a new instance or an instance already managed by the persistence context

EntityManager operations (6)

- `void clear()`:
  - Clear the persistence context. All the objects managed by the `EntityManager` will became detached
- `void flush()`:
  - Synchronize the persistence context with the database

Cascading persistence operations (1)

- By default persistence operations are done only on the entity given as a parameter
  - The operation will not be executed on the other entities connected to the entity
  - For annotations describing logical connections (`@ManyToMany`, `@ManyToOne`, `@OneToMany`, `@OneToOne`) the `cascade` element can be given. With it this default behavior can be overwritten.

Cascade Cascading persistence operations (2)

- Instances of the `javax.persistence.CascadeType` enum tell which operations are to be cascaded:
  - `ALL`
  - `DETACH`
  - `MERGE`
  - `PERSIST`
  - `REFRESH`
  - `REMOVE`

Cascading persistence operations (3)

- Example:

```
@Entity
public class Employee {

    // ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
    // ...

}
```

Cascading persistence operations (4)

- For the `persist()` operation (but only for that!) global cascading can be set in the XML descriptor:

```
<entity-mappings
        xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        version="2.1">
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <cascade-persist/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

Detached entities (1)

- A detached entity is an entity that is not a part of the persistence context
  - Operations done on a detached entity are not saved to the database
- An entity can became detached in several ways e.g.:
  - When closing a persistence context all managed entities will become detached
  - `EntityManager` `clear()` detaches all the managed entities
  - `EntityManager` `detach()` detaches the entity given as a parameter
  - …

Detached entities (2)

- Managing a detached entity (results not in an expected result):
  - Committing the transaction does not change the entity in the database

```
EntityManager em;

Employee emp; // reference to a detached entity

em.getTransaction().begin();
em.merge(emp);
emp.setLastAccessTime(java.time.Instant.now());
em.getTransaction().commit();
```

Detached entities (3)

- Managing a detached entity:
  - Committing the transaction does change the entity in the database

```
EntityManager em;

Employee emp; // reference to a detached entity

em.getTransaction().begin();
Employee managedEmp = em.merge(emp);
managedEmp.setLastAccessTime(java.time.Instant.now());
em.getTransaction().commit();
```

Synchronize with the database

- When a persistence server executes SQL statements in the Db through a JDBC connection we say that the persistence context is **flushed**
  - This can be done any time when the persistence provider finds necessary
  - The flushing is guaranteed in two cases:
    - Committing transaction
    - Calling the `flush()` method of the `EntityManager`

Generating primary key (1)

- A primary key can be automatically generated by the use of the `javax.persistence.GeneratedValue` annotation type
  - It is guaranteed only after flushing the persistence context that the generated key can be reached for the application

Generating primary key (2)

- Four different generation strategies are available. These are represented by instances of the `javax.persistence.GenerationType` enum:
  - **`AUTO`**: Select a proper generating strategy automatically for the given database
  - **`IDENTITY`**: The persistence server uses an *identity* column to generate the primary key
  - **`SEQUENCE`**: The persistence server uses a sequence to generate the primary key
  - **`TABLE`**: The persistence server uses a table to generate the primary key
    - This is the most flexible and portable solution

Generating primary key (3)

- Example:

```
package pkg.model;

import javax.persistence.*;

@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    // ...

}
```

Generating primary key (4)

- Example (2):
  - The DDL statement creating the `Employee` database table in case of Java DB:

```
create table Employee (
    id bigint generated by default as identity,
    /* ... */
    primary key (id)
)
```

Queries (1)

- Java Persistence Query Language (JPQL):
  - A platform independent object oriented query language specified as a part of JPA. In JPQL queries can be formed about persistent entities stored in the database
  - Can be compiled to a target language e.g. SQL
  - 3 different statements: `SELECT`, `UPDATE`, `DELETE`

Queries (2)

- Static and dynamic queries:
  - Defining static queries in annotations or in the XML descriptor files
    - They have names. They are also called named queries. They can be referred by their name when they are executed.
  - Dynamic queries are defined in runtime

Queries (3)

- Queries are represented by the `javax.persistence.Query` and `javax.persistence.TypedQuery<X>` interfaces
  - They can be created by the `createNamedQuery()`, `createNativeQuery()` and `createQuery()` methods of the `EntityManager`

Queries (4)

- Examples of SELECT queries:
  - `SELECT e`
    `FROM Employee e`
  - `SELECT e.name`
    `FROM Employee e`
  - `SELECT e.name, e.salary`
  - `FROM Employee e`
  - `ORDER BY e.name`
  - `SELECT e FROM`
  - `Employee e`
  - `WHERE e.address.state IN ('NY','CA')`

Queries (5)

- Examples of SELECT queries (2):
  - ```
    SELECT d
    FROM Department d
    WHERE SIZE(d.employees) = 2
    ```
  - ```
    SELECT d, COUNT(e), MAX(e.salary),
        AVG(e.salary)
    FROM Department d JOIN d.employees e
    GROUP BY d
    HAVING COUNT(e) >= 5
    ```

Queries (6)

- Example of the execution of dynamic queries

```
TypedQuery<Employee> query = em.createQuery("SELECT e FROM
        Employee e", Employee.class);
List<Employee> emps = query.getResultList();
```

Queries (7)

- Example use of dynamic queries:

```
public class QueryService {

    protected EntityManager em;

    public QueryService(EntityManager em) {
        this.em = em;
    }

    public long queryEmpSalary(String deptName,
            String empName) {
        return em.createQuery("SELECT e.salary "
                + "FROM Employee e "
                + "WHERE e.department.name = :deptName"
                + "    AND e.name = :empName", Long.class)
            .setParameter("deptName", deptName)
            .setParameter("empName", empName)
            .getSingleResult();
    }

    // ...
}
```

Queries (8)

- Example use of static queries:

```
package pkg.model;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQuery(name="Employee.findByName",
    query="SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {

   // ...

}
```

Queries (9)

- Example use of static queries (2):

```
public class QueryService {

    protected EntityManager em;

    public QueryService(EntityManager em) {
        this.em = em;
    }

    // ...

    public Employee findEmployeeByName(String name) {
        return em.createNamedQuery("Employee.findByName",
                Employee.class)
            .setParameter("name", name)
            .getSingleResult();
    }

}
```

Queries (10)

- Criteria API:
  - API to create queries
  - Contained by `javax.persistence.criteria` package

Queries (11)

- Example use of Criteria API:
  - Execution of
    `SELECT e FROM Employee e WHERE e.name = 'John Smith'`

```
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

// ...

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp)
   .where(cb.equal(emp.get("name"), "John Smith"));

TypedQuery<Employee> q = em.createQuery(cq);
Employee emp = q.getSingleResult();
```

Relations (1)

- Relation means a relationship between two entities
  - An entity can be a part of many different relations
- Relation properties:
  - Direction: one direction, two direction (bidirectional)
    - All two directional relations are handled as two one directional relations
  - Multiplicity: `0..1`, `1`, `*`

Relations (2)

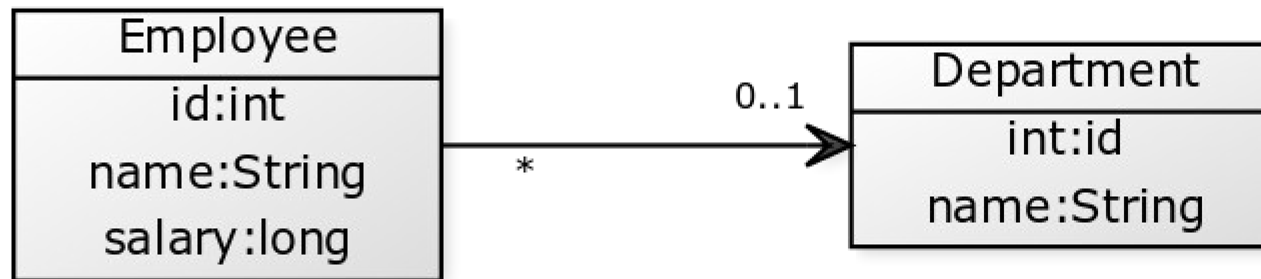
- The following types of relations can be used:
  - *Many-to-one*
  - *One-to-one*
  - *One-to-many*
  - *Many-to-many*

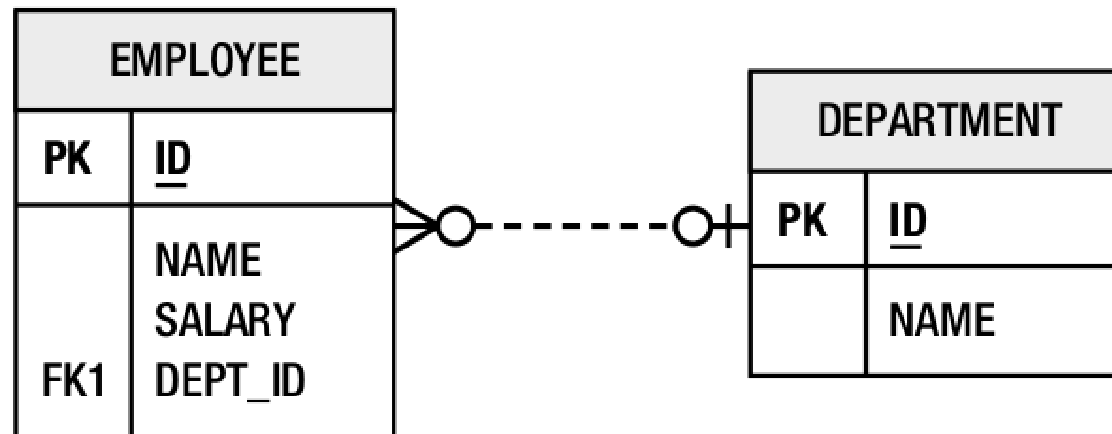
Relations (3)

- *Single-valued association*:
  - Many-to-one
  - One-to-one
    - The bidirectional form is a special case
- *Collection-valued association*:
  - One-to-many
    - The one directional form is a special case
  - Many-to-many

Many-to-one relation (1)

- Example:





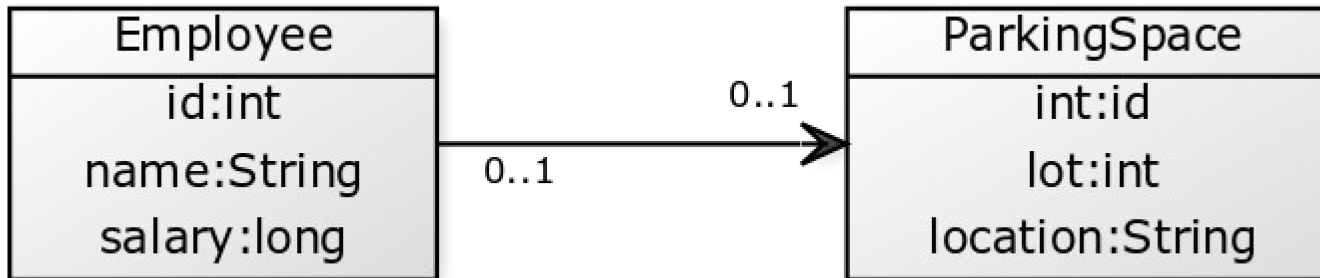
Many-to-one relation (2)

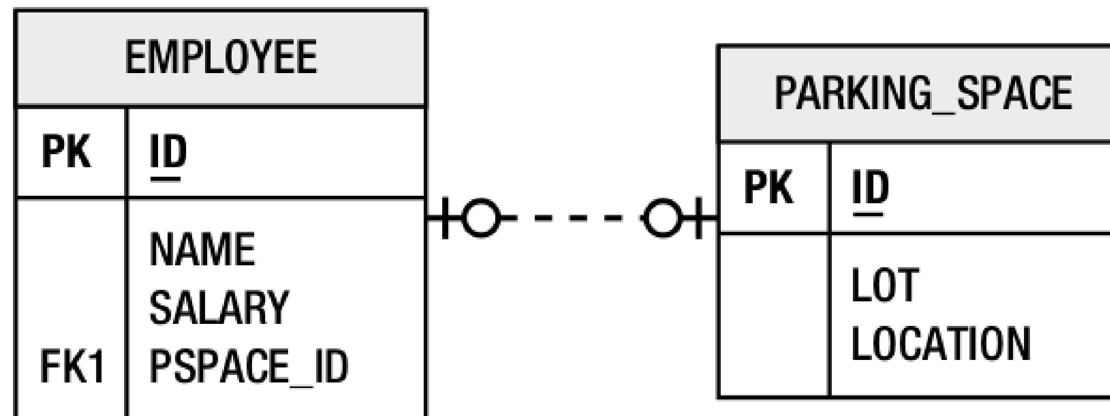
- Example (2):

```
@Entity
public class Employee {

    @Id
    private int id;

    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;

    // ...
}
```

One-to-one relation (1)

- Example:





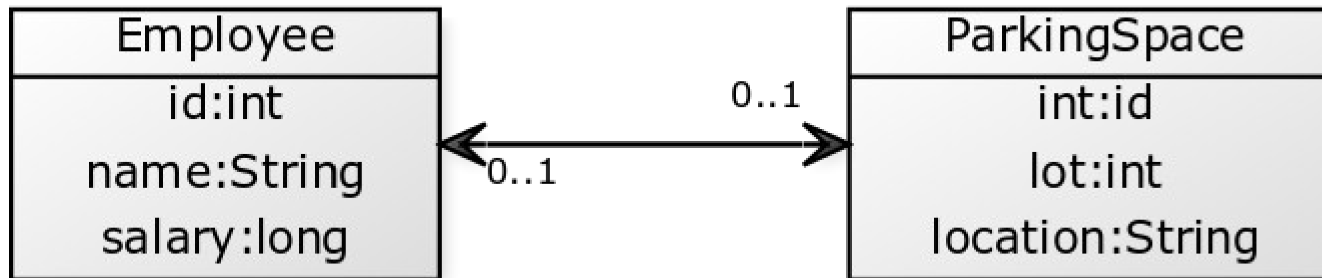
One-to-one relation (2)

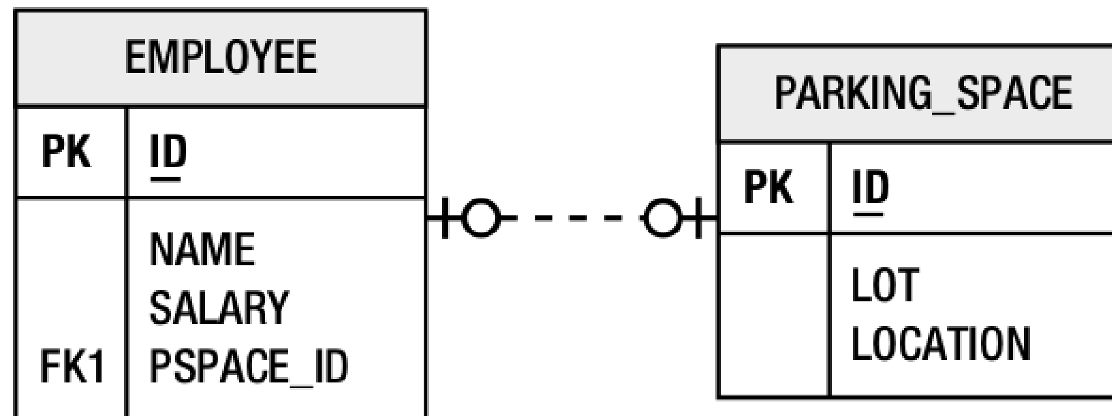
- Example (2):

```
@Entity
public class Employee {

    @Id
    private int id;

    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;

    // ...
}
```

Bidirectional one-to-one relation (1)

- Example:





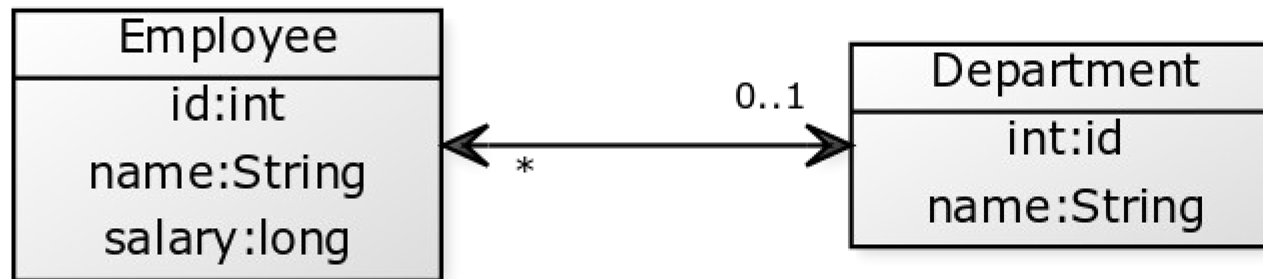
Bidirectional one-to-one relation (2)

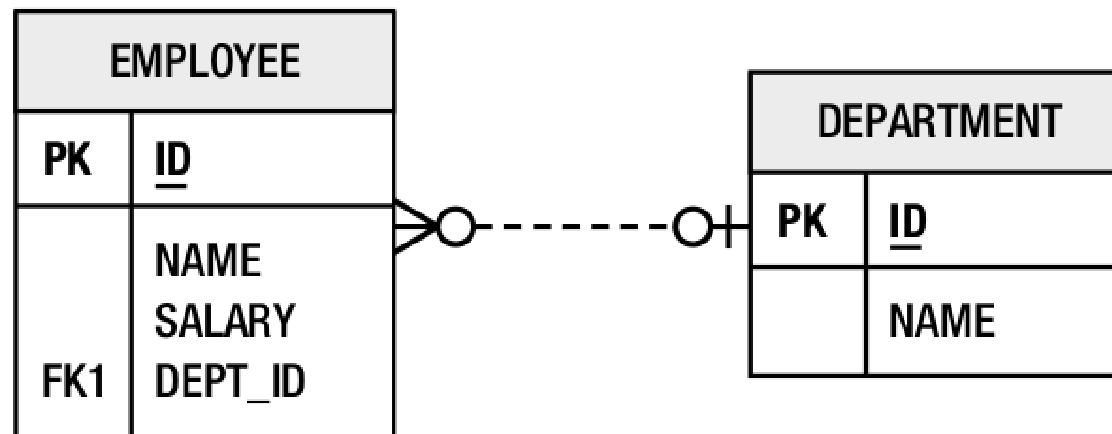
- Example (2):

```
@Entity
public class ParkingSpace {

    @Id
    private int id;

    private int lot;
    private String location;

    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;

    // ...
}
```

One-to many relation (1)

- Example:





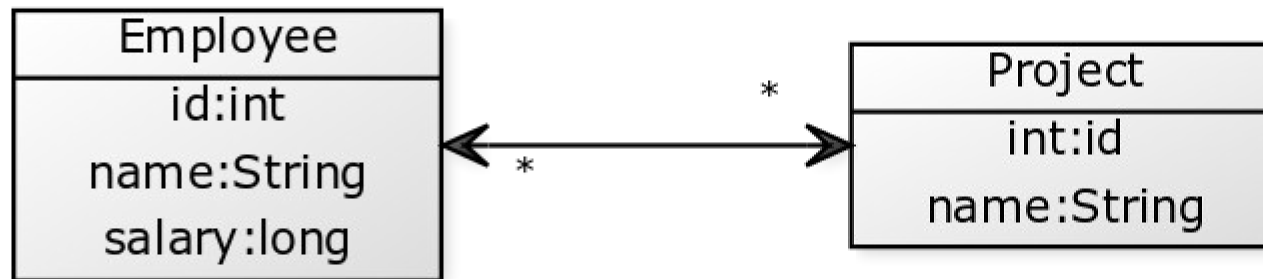
One-to many relation (2)

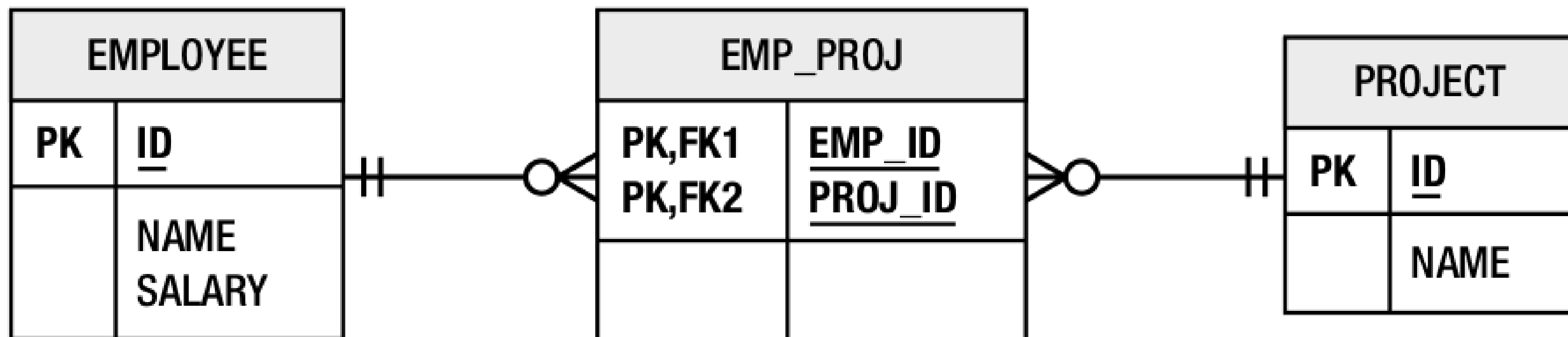
- Example (2):

```
@Entity
public class Department {

    @Id
    private int id;

    private String name;

    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;

    // ...
}
```

Many-to-many relation (1)

- Example:





Many-to-many relation (2)

- Example (2):

```
@Entity
public class Employee {

    @Id
    private int id;

    private String name;

    @ManyToMany
    @JoinTable(name="EMP_PROJ",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;

    // ...
}
```

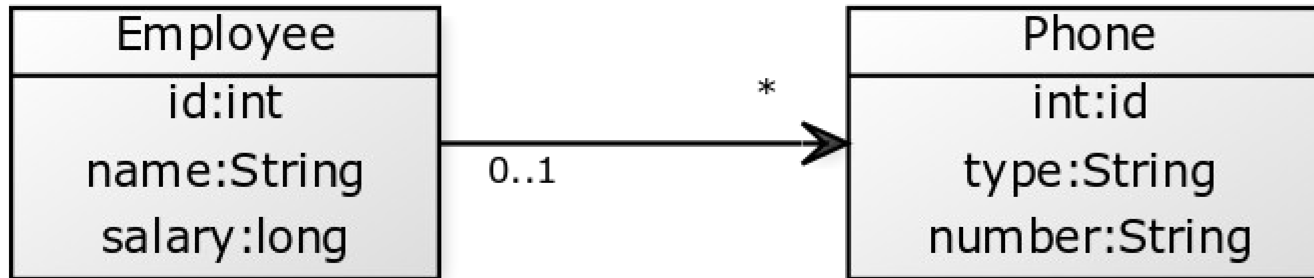
Many-to-many relation (3)

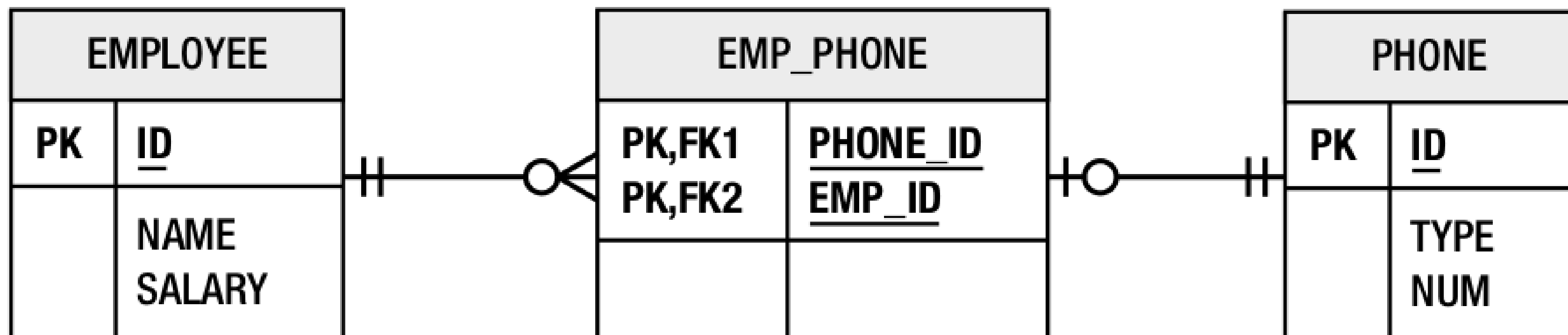
- Example (3):

```
@Entity
public class Project {

    @Id
    private int id;

    private String name;

    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;

    // ...
}
```

One directinal
One-to-many relation (1)

- Example:





One directinal
One-to-many relation (2)

- Example:

```
@Entity
public class Employee {

    @Id
    private long id;

    private String name;

    @OneToMany
    @JoinTable(name="EMP_PHONE",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;

    // ...
}
```

ElementCollections (1)

- ElementCollections are colections of embeddable objects or other basic type elements
  - Basic types e.g.: `java.lang.String`, `java.lang.Integer`, `java.math.BigDecimal`, ...
- They do not describe relations
  - Relations are always between independent entities. Elements of ElementCollections however can be reached only through the containing entity.

ElementCollections (2)

- Example:

```
@Embeddable
public class VacationEntry {

    private LocalDate startDate;

    @Column(name="DAYS")
    private int daysTaken;

    // ...
}
```

ElementCollections (3)

- Example (2):

```
@Entity
public class Employee {

    @Id
    private int id;

    private String name;
    private long salary;

    @ElementCollection(targetClass=VacationEntry.class)
    private Collection vacationBookings;

    @ElementCollection
    private Set<String> nickNames;

    // ...
}
```

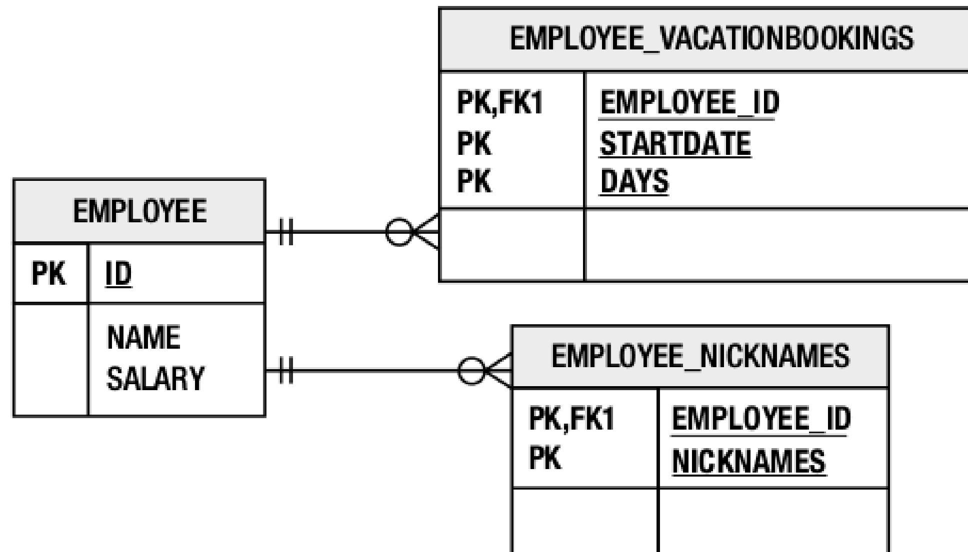
ElementCollections (4)

- Example (3):

```
@Entity
public class Employee {

    @Id
    private int id;
    private String name;
    private long salary;

    @ElementCollection(targetClass=VacationEntry.class)
    @CollectionTable(name="VACATION",
        joinColumns=@JoinColumn(name="EMP_ID"))
    @AttributeOverride(name="daysTaken",
        column=@Column(name="DAYS_ABS"))
    private Collection vacationBookings;

    @ElementCollection
    @Column(name="NICKNAME")
    private Set<String> nickNames;

    // ...
}
```

ElementCollections (5)

- Example (4):



ElementCollections (6)

- Example use of lists:

```
@Entity
public class Employee {

    @Id
    private int id;
    private String name;
    private long salary;

    @ElementCollection
    @Column(name="NICKNAME")
    @OrderBy
    private List<String> nickNames;

    // ...
}
```

ElementCollections (7)

- Example use of lists:

```
@Entity
public class Employee {

    @Id
    private int id;
    private String name;
    private long salary;

    @ElementCollection
    @Column(name="NICKNAME")
    @OrderColumn(name="NICKNAME_INDEX")
    private List<String> nickNames;

    // ...
}
```

Logging (1)

- Hibernate uses the JBoss Logging framework for logging
  https://github.com/jboss-logging/jboss-logging
  - Several logging frameworks can also be plugged e.g.: `java.util.logging` package or SLF4J
- See:
  - Thorben Janssen. *Hibernate Logging Guide – Use the right config for development and production*. December, 2015.
    http://www.thoughts-on-java.org/hibernate-logging-guide/

Logging (2)

- Most important logging categories:

Category	Description
`org.hibernate`	Contains all the log messages
`org.hibernate.SQL`	Executed SQL statements
`org.hibernate.type.descriptor.sql`	Parameters of SQL statements
`org.hibernate.tool.hbm2ddl`	Executed DDL statements

Logging (3)

- Formatting of SQL statements can be set in the `persistence.xml` fájl in the following way:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        version="2.1">
    <persistence-unit name="...">
        <!-- ... -->
        <properties>
            <!-- ... -->
            <property name="hibernate.format_sql"
                value="true"/>
            <property name="hibernate.use_sql_comments"
                value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Further reading

- Christian Bauer, Gavin King, Gary Gregory. *Java Persistence with Hibernate*. 2nd edition. Manning, 2015. https://www.manning.com/books/java-persistence-with-hibernate-second-edition
- Mike Keith, Merrick Schincariol. *Pro JPA 2 – A Definitive Guide to Mastering the Java Persistence API*. 2nd edition. Apress, 2013. http://www.apress.com/9781430249269
- Yogesh Prajapati, Vishal Ranapariya. *Java Hibernate Cookbook – Over 50 recipes to help you build dynamic and powerful real-time Java Hibernate applications*. Packt Publishing, 2015.
- James Sutherland, Doug Clarke. *Java Persistence*. Wikibooks.org, 2013. https://en.wikibooks.org/wiki/Java_Persistence