laboratory Gerstner

Persisting Data, ORM

Petr Křemen

petr.kremen@fel.cvut.cz

What „data persistence“ means ?

- we manipulate data (represented as object state) that need to be stored
  - ***persistently***
    - to survive a single run of the application
  - ***queriably***
    - to be able to retrieve/access them
  - ***scalably***
    - to be able to handle large data volumes
  - ***reliably***, ***transactionally, ...***

How to achieve persistence

- **Serialization**
  - simple, yet hardly queriable, not transactional, …
  - stream persisting an instance of class `C` is deprecated once definition of `C` is modified (e.g. field added/removed).
- **Relational Databases** ***(MySQL, PostgreSQL, Oracle, ...)***
  - efficient storage for **data with rigid schema**
  - well-established and most popular technology
  - efficient search using SQL standard
  - secure and Transactional (ACID)

How to achieve persistence (cont.)

- **NoSQL databases**
  - Key-value storages *(MongoDB, Hadoop, …)*
    - suitable for **data without rigid schema**
  - Object Databases
    - designed in 90's to capture complexity of object models (e.g. inheritance)
    - Issues: scalability, standardized queries
  - Triple Stores *(SDB, TDB, Sesame, Virtuoso, ...)*
    - graph stores
    - for distributed semantic web data – RDF(S), OWL

Programmatic Access to Relational Databases (RDBMS)

- JDBC (JSR 221)
  - Java standard to ensure independence on the particular RDBMS (at least theoretically)
- EJB 2.1 (JSR 153)
  - Provides Object Relational Mapping (ORM), but complicated (single entity = several Java files + XMLs)
  - distributed transactions, load balancing
- iBatis,Hibernate – ORM driving forces for JPA 2
- JPA 2 (JSR 317)
  - Standardized ORM solution for both standalone and Java EE applications

JDBC

- Java standard to ensure independence on the particular RDBMS (at least theoretically)

```
Connection connection = null;
PreparedStatement statement = null;
try {
     Class.forName("org.postgresql.Driver");
    connection = DriverManager.getConnection(jdbcURL,dbUser,dbPassword);
    statement =  connection.prepareStatement("SELECT * FROM PERSON WHERE HASNAME LIKE ?");
    statement.setString(1, "%Pepa%");
    ResultSet rs = statement.executeQuery();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
    if ( statement != null ) {
         try {
              statement.close();
         } catch (SQLException e1) { e1.printStackTrace(); }
     }
    if ( connection != null ) {
         try {
              connection.close();
         } catch (SQLException e1) { e1.printStackTrace();}
     }
}
```

JDBC – entities CRUD

Create:

```
PreparedStatement statement =
connection.prepareStatement("INSERT INTO
PERSON (id,hasname) VALUES (?,?)");
statement.setLong(1,10);
statement.setString(2,"Honza");
statement.executeUpdate();
```

Update:

```
PreparedStatement statement =
connection.prepareStatement("UPDATE PERSON
SET HASNAME='Jirka' WHERE ID = ?");
statement.setLong(1,2);
statement.executeUpdate();
```

Retrieve:

```
PreparedStatement statement =
connection.prepareStatement("SELECT *
FROM PERSON WHERE ID = ?");
statement.setLong(1,2);
ResultSet rs = statement.executeQuery();
```

Delete:

```
PreparedStatement statement =
connection.prepareStatement("DELETE FROM
PERSON WHERE ID=?");
statement.setLong(1,1);
statement.executeUpdate();
```

Question 1: Why prepared statements ?

How to avoid the boilerplate code

- Boilerplate code
  - Obtaining (pooled) connection
  - `SQLException` handling
  - creating Java objects out of the query results:

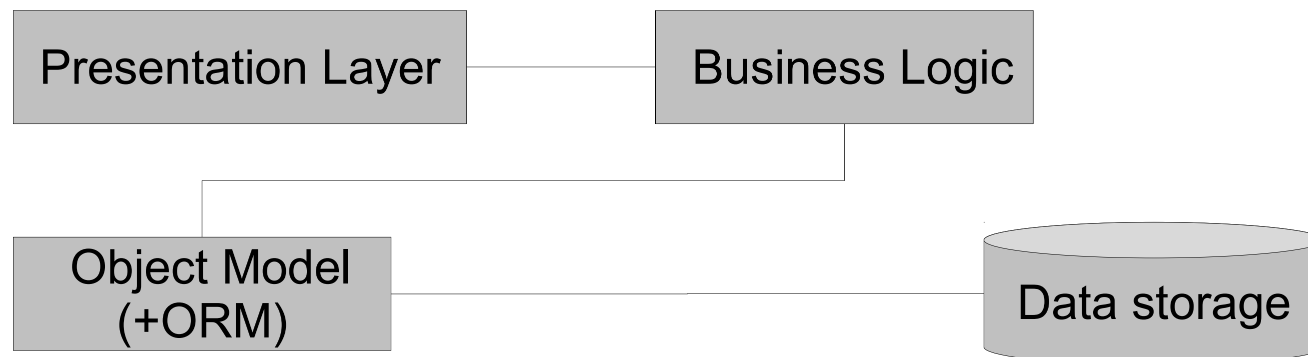
```
ResultSet rs = …
while(rs.next()) {
   Person p = new Person();
   p.setId(rs.getLong("ID"));
   p.setHasName(rs.getString("HASNAME"));
}
```

- Although SQL is a standard – there are still differences in implementations (MySQL autoincrement, PostgreSQL serial ...)

solution = **O**bject **R**elational **M**apping (ORM)

ORM architecture

- idea: „map whole Java classes to database records“
- a typical system architecture with ORM:




```
@Entity
public Person {
    @Id
    private Long id;
   private String hasName;
    // setters+getters
}
```

```
PERSON
================================
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)
```

CRUD using JPA 2.0

Initialization:

```
EntityManagerFactory f = Persistence.createEntityManagerFactory("pu");
EntityManager em = f.createEntityManager();
EntityTransaction t = em.getTransaction();
t.begin();
```

Create:

```
Person person = new Person();
person.setId(10);
Person.setHasName("Honza");
em.persist(person);
```

Retrieve:

```
Person person = em.find(Person.class, 2);
```

Update:

```
Person person = em.find(Person.class, 2);
person.setHasName(“Jirka“);
```

Delete:

```
Person person = em.find(Person.class, 1);
em.remove(person);
```

Finalization:

```
t.commit();
```

JPA 2.0

- Java Persistence API 2.0 (JSR-317)
- Although part of Java EE 6 specifications, JPA 2.0 can be used both in EE and SE applications.
- Main topics covered:
  - Basic scenarios
  - Controller logic – `EntityManager` interface
  - ORM strategies
  - JPQL + Criteria API

JPA 2.0 – Entity Example

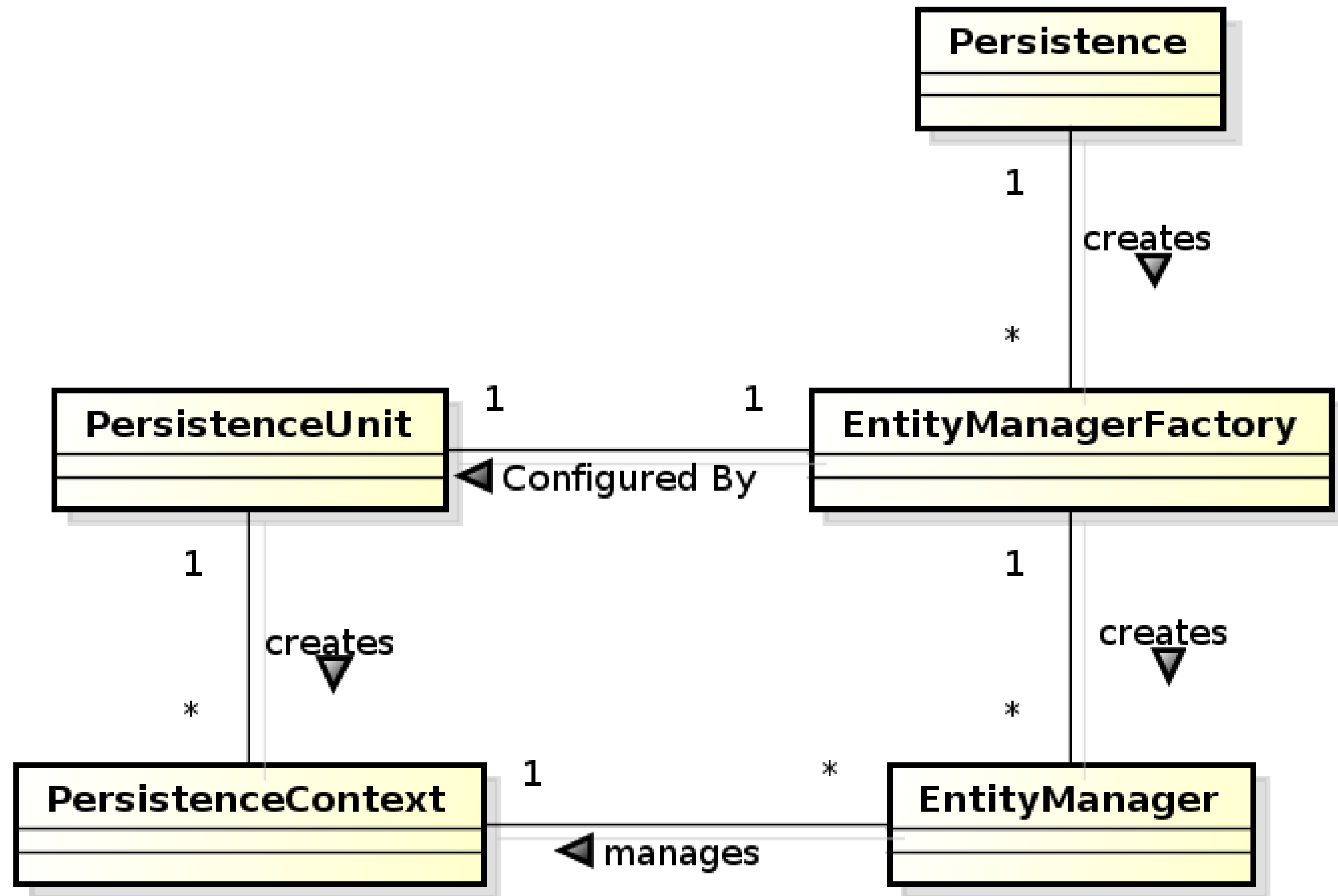
- Minimal example (configuration by exception):

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    // setters + getters
}
```

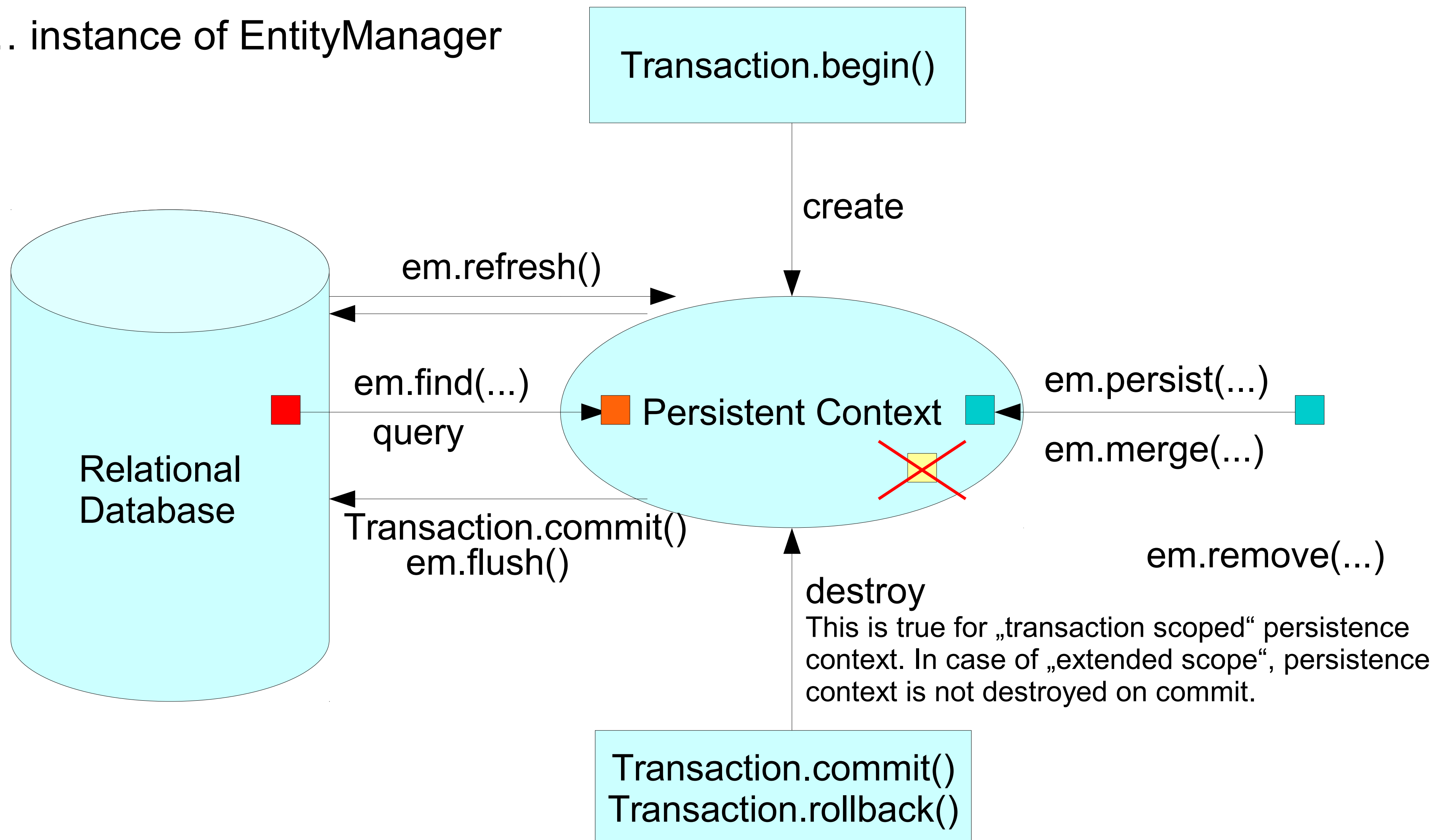
JPA 2.0 - Basics

- Let's have a set of „suitably annotated“ POJOs, called ***entities***, describing your domain model.
- A set of entities is logically grouped into a ***persistence unit***.
- JPA 2.0 providers :
  - generate persistence unit from existing database,
  - generate database schema from existing persistence unit.
- What is the benefit of the keeping Your domain model in the persistence unit entities (OO) instead of the database schema (SQL)

JPA 2.0 – Basic concepts

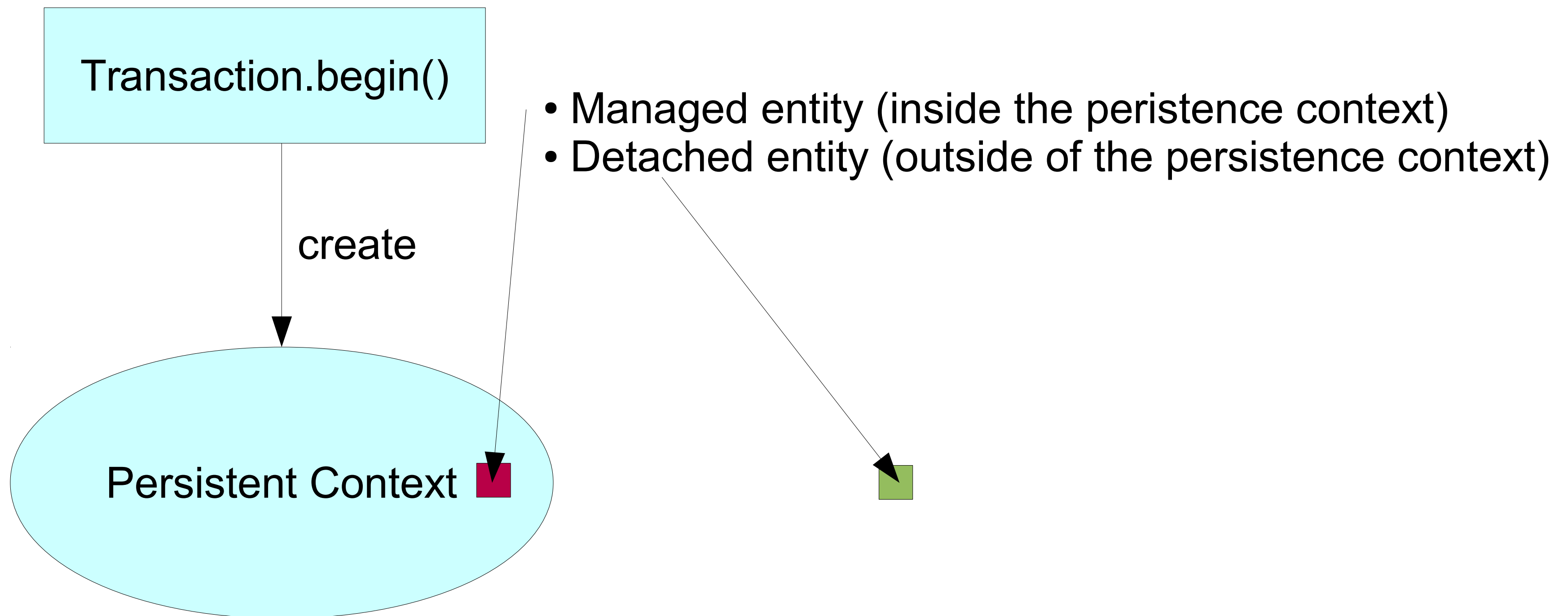


JPA 2.0 – Persistence Context



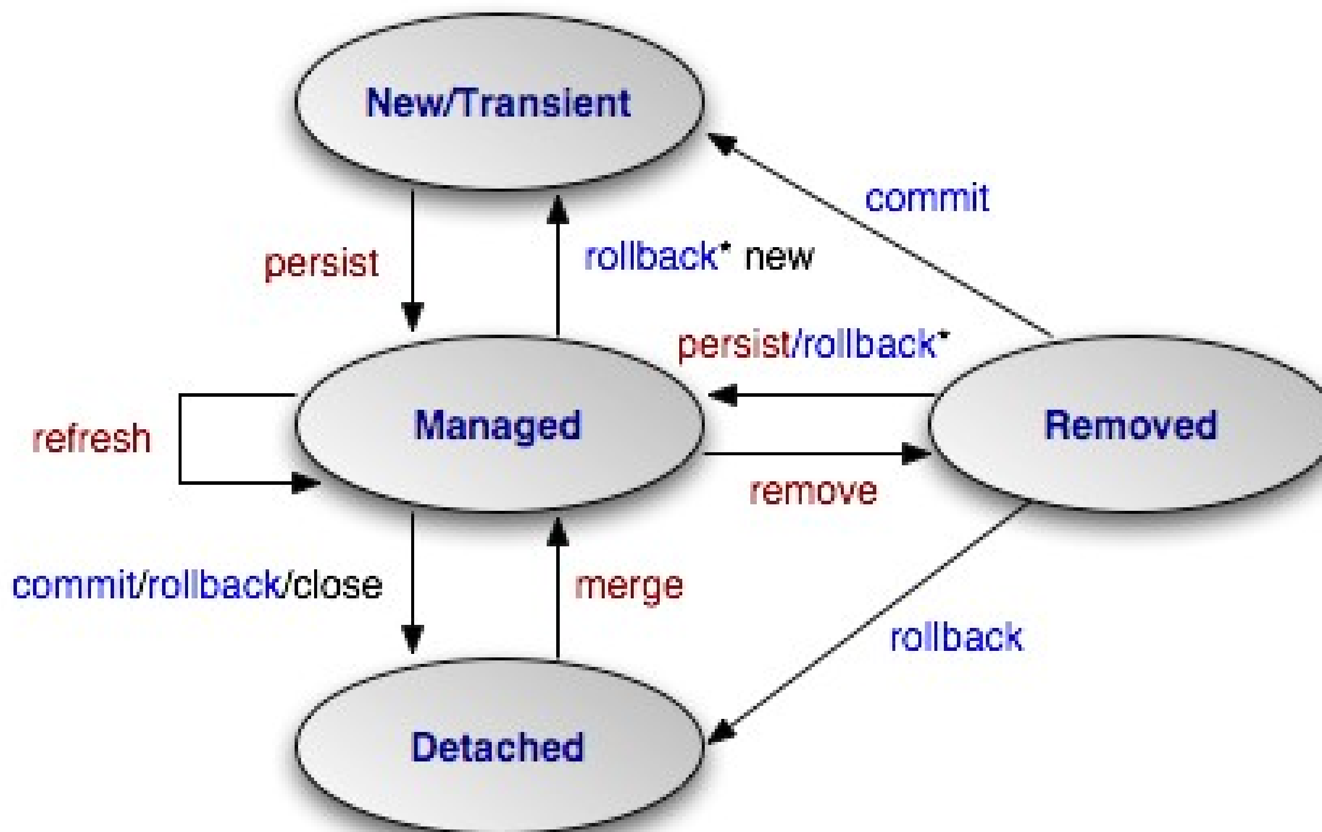
JPA 2.0 – Persistence Context

em … instance of EntityManager



- em.persist(entity) … persistence context must not contain an entity with the same id
- em.merge(entity) … merging the state of an entity existing inside the persistence context and its other incarnation outside

Entity Lifecycle



(C) Wikipedia, http://cs.wikipedia.org/wiki/Java_Persistence_API

JPA 2.0 – Persistence Context

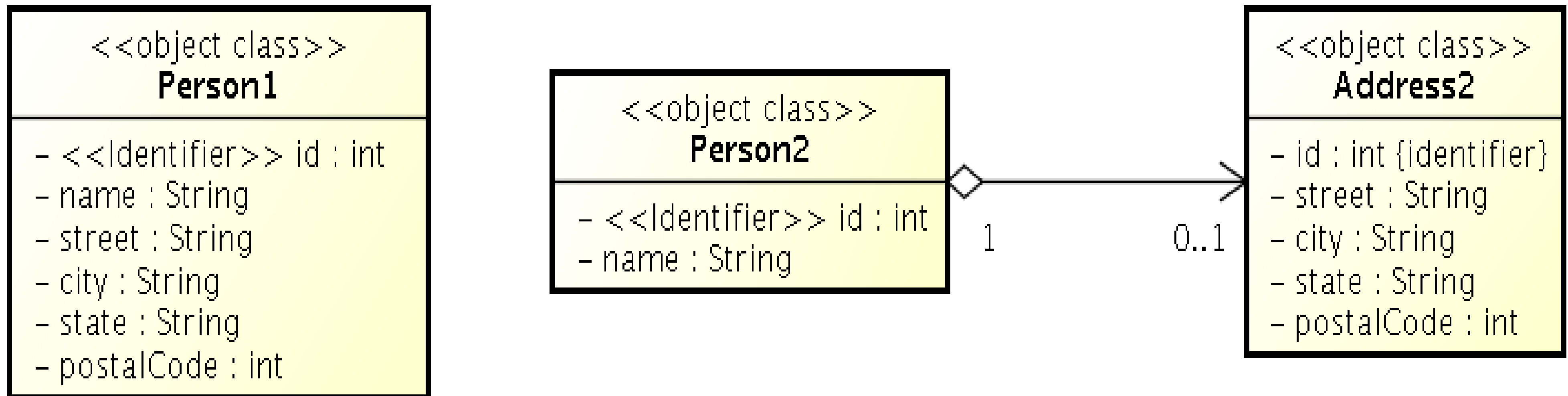
- In runtime, the application accesses the object counterpart (represented by entity instances) of the database data. These (*managed*) entities comprise a ***persistence context (PC)***.
  - PC is synchronized with the database on demand (refresh, flush) or at transaction commit.
  - PC is accessed by an `EntityManager` instance and can be shared by several `EntityManager` instances.

JPA 2.0 – EntityManager

- **`EntityManager` (EM)** instance is in fact a generic DAO, while entities can be understand as DPO (managed) or DTO (detached).
- Selected operations on EM (CRUD) :
  - ***C**reate* : `em.persist(Object o)`
  - ***R**ead* : `em.find(Object id), em.refresh(Object o)`
  - ***U**pdate* : `em.merge(Object o)`
  - ***D**elete* : `em.remove(Object o)`
  - native/JPQL queries: `em.createNativeQuery`, `em.createQuery`, etc.
  - Resource-local transactions: `em.getTransaction.[begin(),commit(),rollback()]`

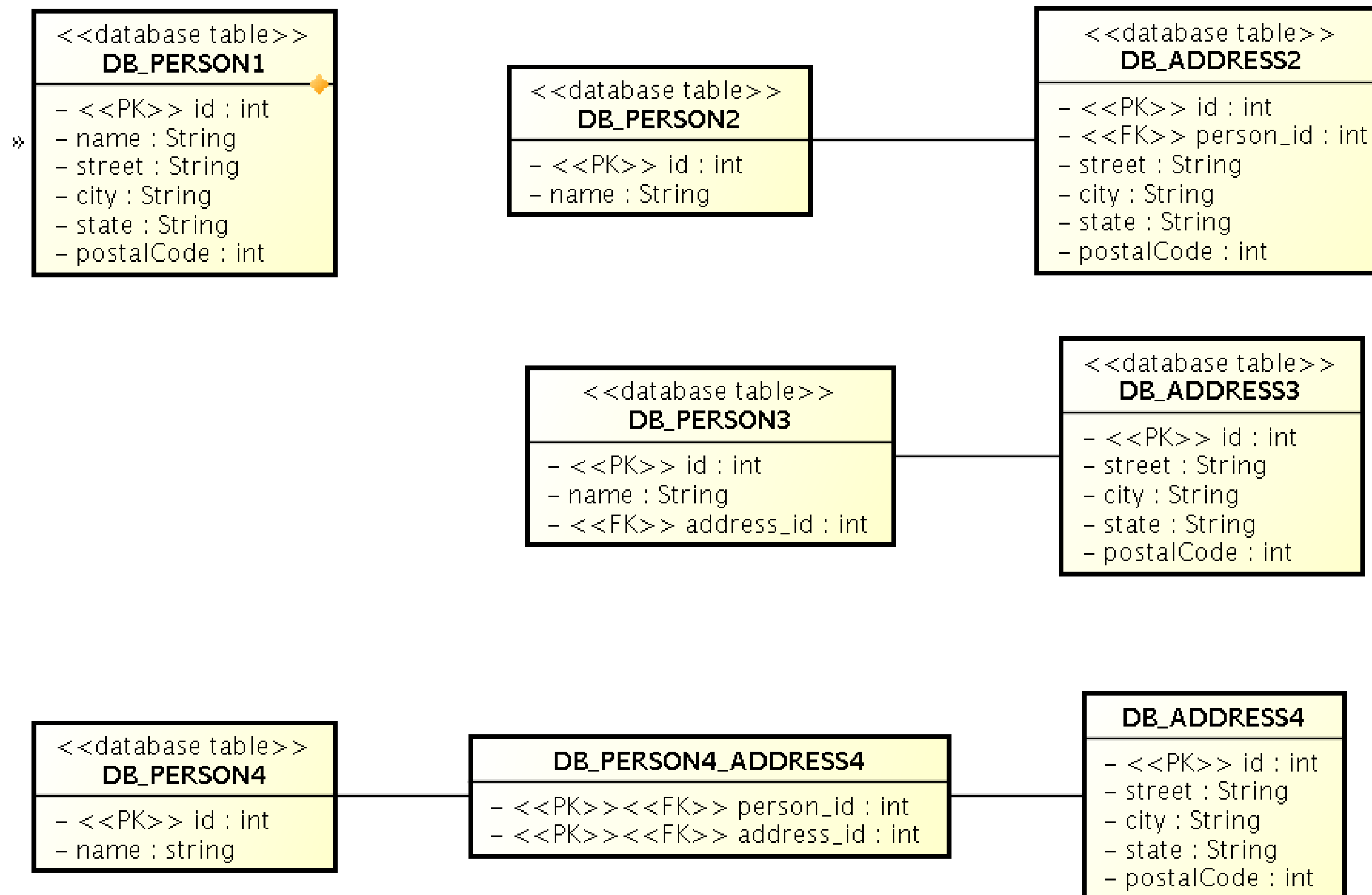
Example – object model

- When would You stick to one of these options ?



Example – database

- … and how to model it in SQL ?



ORM - Basics

- Simple View
  - Object classes = entities = SQL tables
  - Object properties (fields/accessor methods) = entity properties = SQL columns
- The ORM is realized by means of Java annotations/XML.
- Physical Schema annotations
  - @Table, @Column, @JoinColumn, @JoinTable, etc.
- Logical Schema annotations
  - @Entity, @OneToMany, @ManyToMany, etc.
- Each property can be fetched lazily/eagerly.

ORM – Basic data types

- Primitive Java types: String → varchar/text, Integer → int, Date → TimeStamp/Time/Date, etc.
- Wrapper classes, basic type arrays, Strings, temporal types
- @Column – physical schema properties of the particular column (insertable, updatable, precise data type, defaults, etc.)
- @Lob – large objects
- Default EAGER fetching (except Lobs)

```
@Column(name="id")
private String getName();
```

ORM – Enums, dates

- `@Enumerated(value=EnumType.String)`

  `private EnumPersonType type;`

  - Stored either in a text column, or in an int column
- `@Temporal(TemporalType.Date)`

  `private java.util.Date datum;`

  - Stored in respective column type according to the TemporalType.

ORM – Identifiers

- Single-attribute: @Id,
- Multiple-attribute – an identifier class must exist
  - Id. class: @IdClass, entity ids: @Id
  - Id. class: @Embeddable, entity id: @EmbeddedId
- How to write `hashCode`, `equals` for entities ?
-
```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;
```

Generated Identifiers

Strategies

- AUTO - the provider picks its own strategy
- TABLE – special table keeps the last generated values
- SEQUENCE – using the database native SEQUENCE functionality (PostgreSQL)
- IDENTITY – some DBMSs implement autonumber column

For database-related strategies, the value of id is set only on
- commit
- em.flush()
- em.refresh()

Generated Identifiers TABLE strategy

```
@TableGenerator(
    name="Address_Gen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    pkColumnName="AddrGen",
    initialValue=10000,
    allocationSize=100)
@Id @GeneratedValue(generator="AddressGen")
private int id;
```

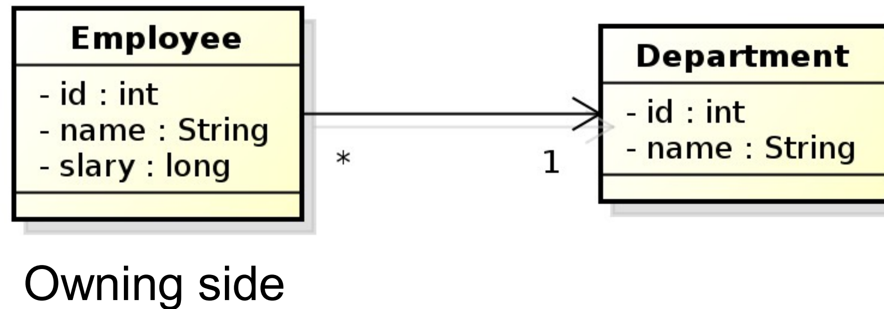
ORM – Relationships

- Unidirectional vs. Bidirectional
- `@OneToMany`
  - Forgotten `mappedBy`
- `@ManyToOne`
- `@ManyToMany`
  - Two `ManyToMany` relationships from two different entities
- `@OneToOne`
- `@JoinColumn, @JoinTable` – in the owning entity (holding the foreign key)
- Cascading

ORM – Relationships

		unidirectional	bidirectional
many-to-one	owning	@ManyToOne [@JoinColumn]	@ManyToOne [@JoinColumn]
	inverse	X	@OneToMany(mappedBy)
one-to-many	owning	@OneToMany [@JoinTable]	@ManyToOne [@JoinColumn]
	inverse	X	@OneToMany(mappedBy)
one-to-one	owning (any)	@OneToOne [@JoinColumn]	@OneToOne [@JoinColumn]
	inverse (the other)	X	@OneToOne(mappedBy)
many-to-many	owning (any)	@ManyToMany [@JoinTable]	@ManyToMany [@JoinTable]
	inverse (the other)	X	@ManyToMany(mappedBy)

Unidirectional many-to-one relationship

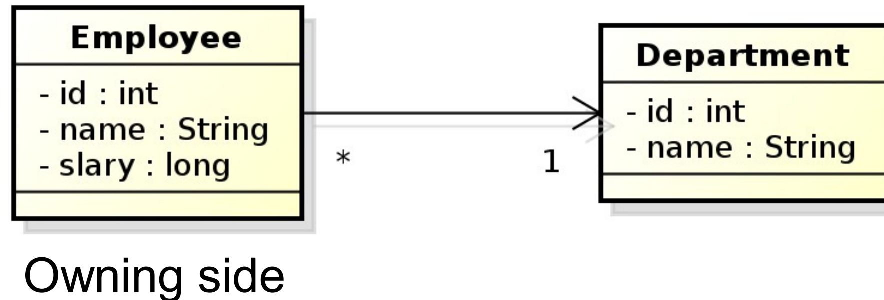


Owning side

```
@Entity
public class Employee {
    // …
    @ManyToOne
    private Department department;
    // …
}
```

In database, the N:1 relationship is implemented by means of a foreign key placed in the Employee table. In this case, the foreign key has a default name.

Unidirectional many-to-one relationship

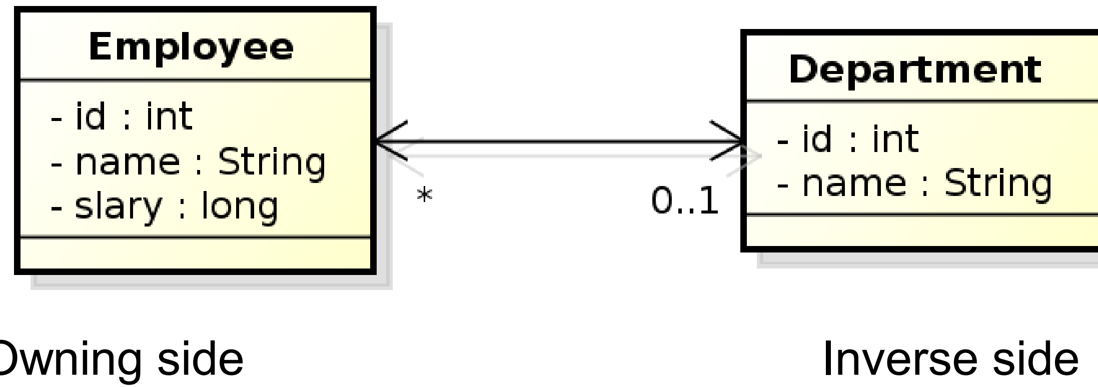


Owning side

```
@Entity
public class Employee {

   @Id private int id;
   Private String name;
   @ManyToOne
   @JoinColumn(name="DEPT_ID")
   private Department department;

}
```

In this case, the foreign key is defined my means of the @JoinColumn annotation.

Bidirectional many-to-one relationship



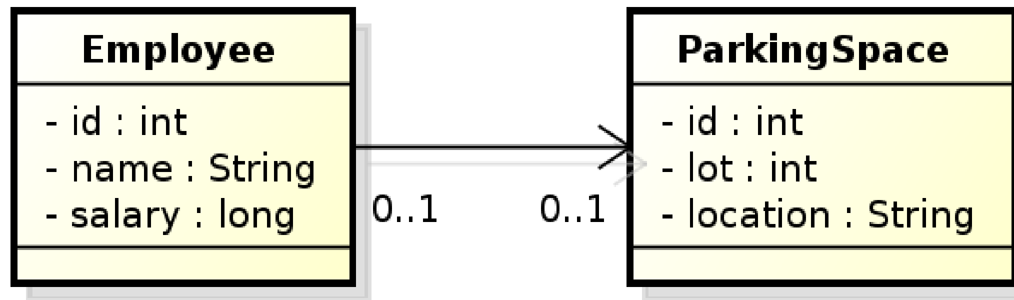
Owning side

Inverse side

```
@Entity
public class Employee {

   @Id private int id;
   private String name;
   @ManyToOne
   @JoinColumn(name="DEPT_ID")
   private Department department;

}
```

```
@Entity
public class Department {

   @Id private int id;
   private String name;
   @OneToMany(mappedBy="department")
   private Collection<Employee>
                                   employees;

}
```

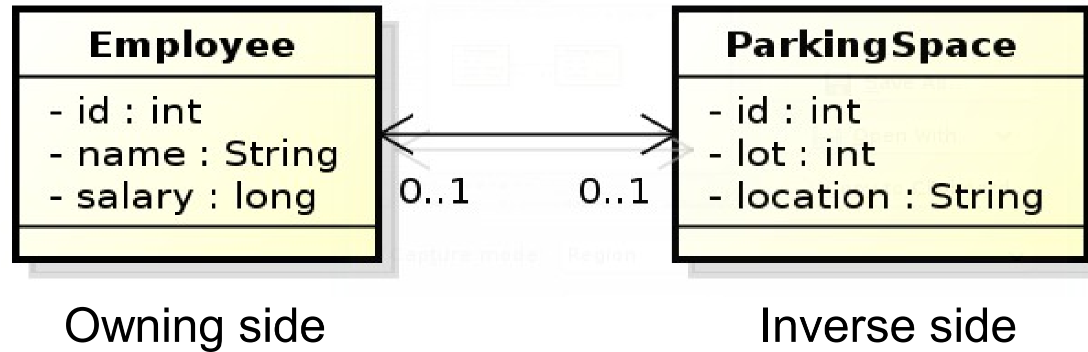
Unidirectional one-to-one relationship



Owning side

```
@Entity
public class Employee {

    @Id private int id;
    private String Name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;

}
```

Bidirectional one-to-one relationship



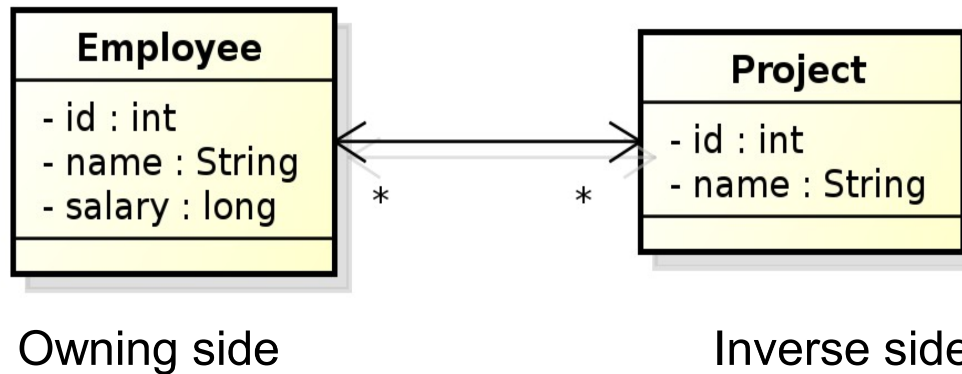
Owning side

Inverse side

```
@Entity
public class Employee {

   @Id private int id;
   private String Name;
   @OneToOne
   @JoinColumn(name="PSPACE_ID")
   private ParkingSpace
                    parkingSpace;

}
```

```
@Entity
public class ParkingSpace {

   @Id private int id;
   private int lot;
   private String location;
   @OneToOne(mappedBy="parkingSpace");
   private Employee
                    employee;

}
```

Bidirectional many-to-many relationship



Owning side

Inverse side

```
@Entity
public class Employee {

   @Id private int id;
   private String Name;
   @ManyToMany
   private Collection<Project>
                              project;

}
```

```
@Entity
public class Project {

   @Id private int id;
   private String name;
   @ManyToMany(mappedBy="projects");
   private Collection<Employee>
                              employees;

}
```

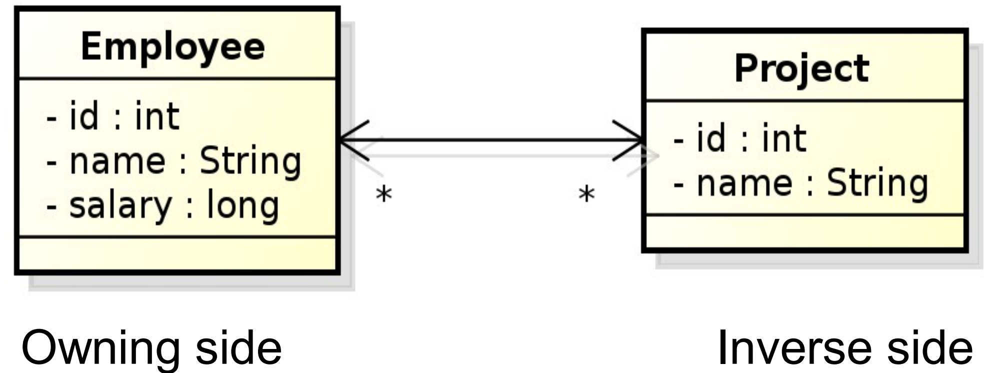
In database, N:M relationship must be implemented by means of a table with two foreign keys. In this case, both the table and its columns have default names.

Conceptual Intermezzo

- Be careful in case of N:M relationships.

- Does it mean that
  - A patient has **one** treatment that are handled in **more** hospital ?
  - A patient has **more** treatments, each handled in a **single** hospital ?
  - …
- partialities and cardinalities do not need to be enough. Careful conceptual modeling often leads to **decomposing M:N relationships on the conceptual level.**

Bidirectional many-to-many relationship



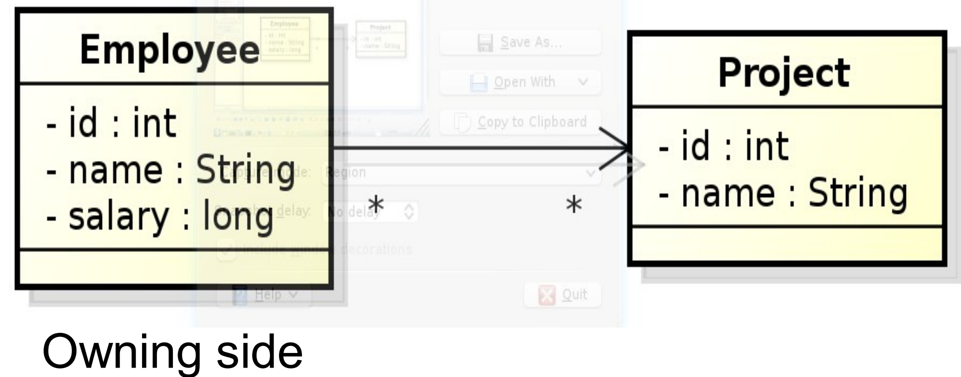
Owning side

Inverse side

```
@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;

}
```

```
@Entity
public class Project {

    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects");
    private Collection<Employee> employees;

}
```

Unidirectional many-to-many relationship

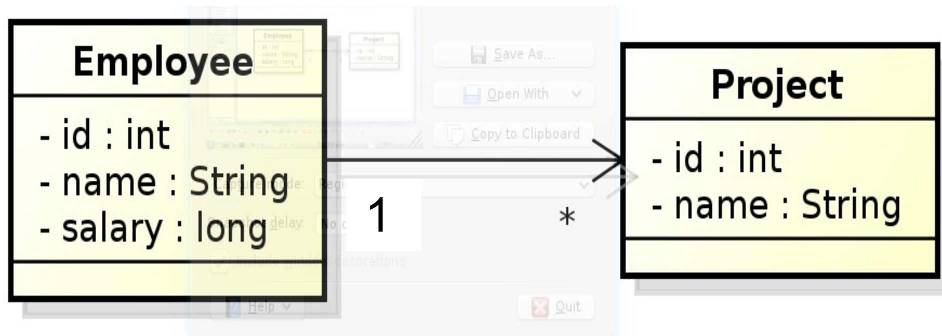


Owning side

```
@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;

}
```

```
@Entity
public class Project {

    @Id private int id;
    private String name;

}
```

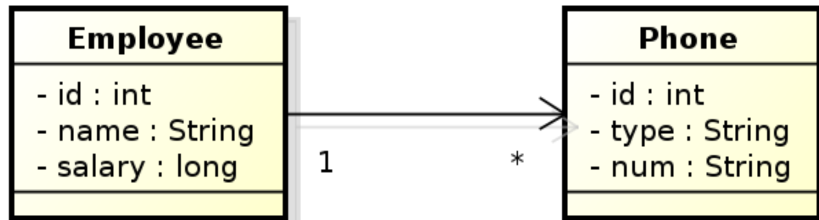
Unidirectional one-to-many relationship



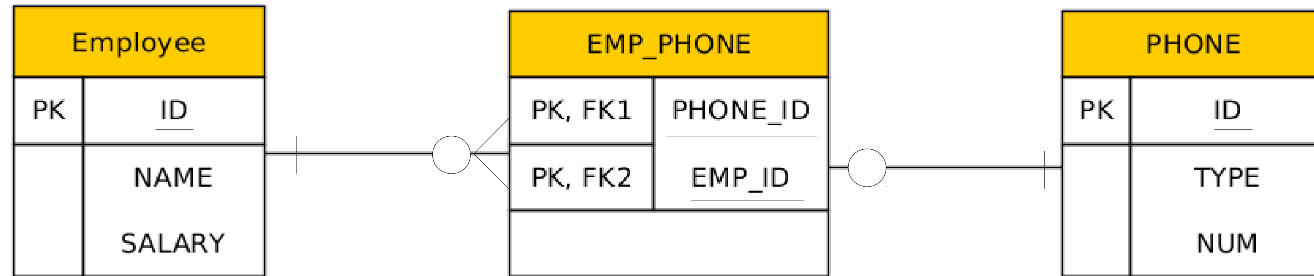
Owning side

```
@Entity
public class Employee {

    @Id private int id;
    private String name;
    @OneToMany
    @JoinTable(name="EMP_PROJ",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;

}
```

Unidirectional one-to-many relationship



Owning side



Logical database schema

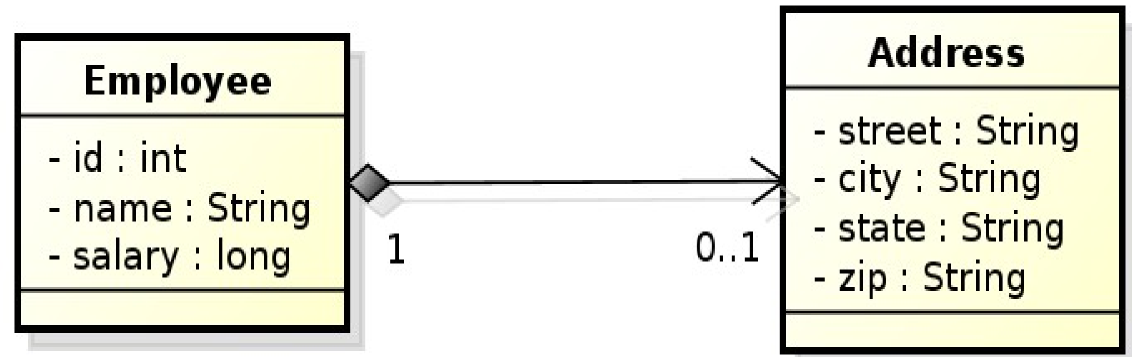
```
@Entity
public class Employee {

    @Id private int id;
    private String name;
    private float salary;
    @OneToMany
    @JoinTable(name="EMP_PHONE",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Project> phones;

}
```

Lazy Relationships

```
@Entity
public class Employee {

   @Id private int id;
   private String name;

   @OneToOne(fetch=FetchType.LAZY)
   private ParkingSpace parkingSpace;

}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	STATE
	ZIP_CODE



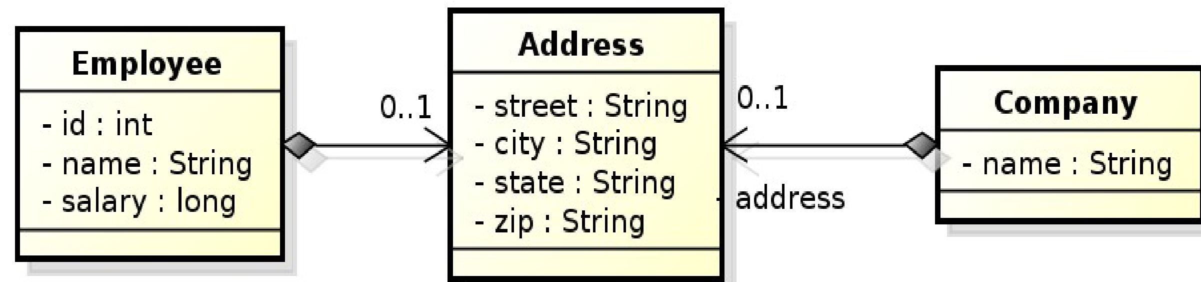

```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
   private String street;
   private String city;
   private String state;
   @Column(name="ZIP_CODE")
   private String zip;
}
```

```
@Entity
public class Employee {
   @Id private int id;
   private String name;
   private long salary;
   @Embedded private Address
                           address;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE

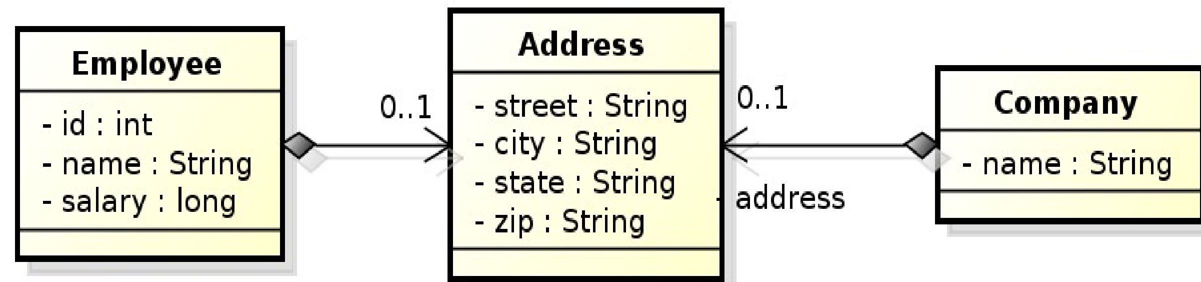



```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE

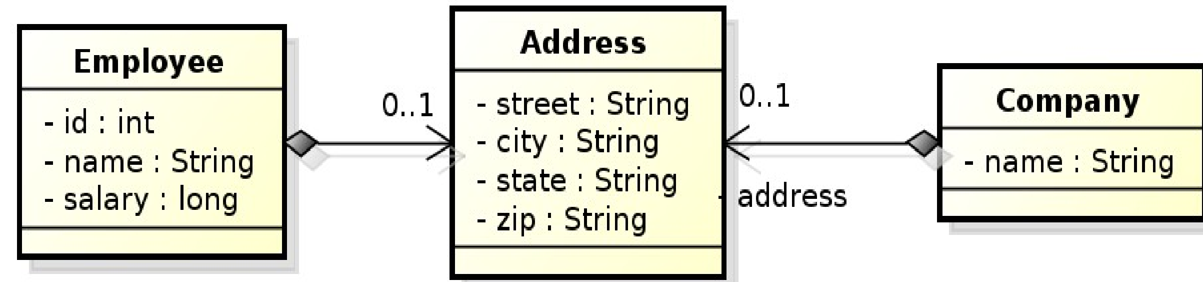



```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name=“state“, column=@Column(name=“PROVINCE“)),
        @AttributeOverride(name=“zip“, column=@Column(name=“POSTAL_CODE“))
    })
    private Address address;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE




```
@Entity
public class Company {
    @Id private String name;
    @Embedded
    private Address address;
}
```

How to map legacy databases

1. One entity to many tables: *@SecondaryTable, @Column(table=...)*

```
@SecondaryTables({
  @SecondaryTable(name="ADDRESS")
})
public class Person {

  @Id
  private Long id;

  @Column(table="ADDRESS")
  private String city;

  // getters + setters
}
```

```
PERSON
================================
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)
```

```
ADDRESS
================================
ID bigint
     PRIMARY KEY NOT NULL
CITY varchar(255)
FOREIGN KEY (id)
     REFERENCES person (id)
```

How to map legacy databases

2. Multiple entities to one table: *@Embedded,@EmbeddedId,@Embeddable*

```java
@Entity
public class Person {
   @Id
   private Long id;
   private String hasName;

   @Embedded
   private Birth birth;
   // getters + setters
}
```

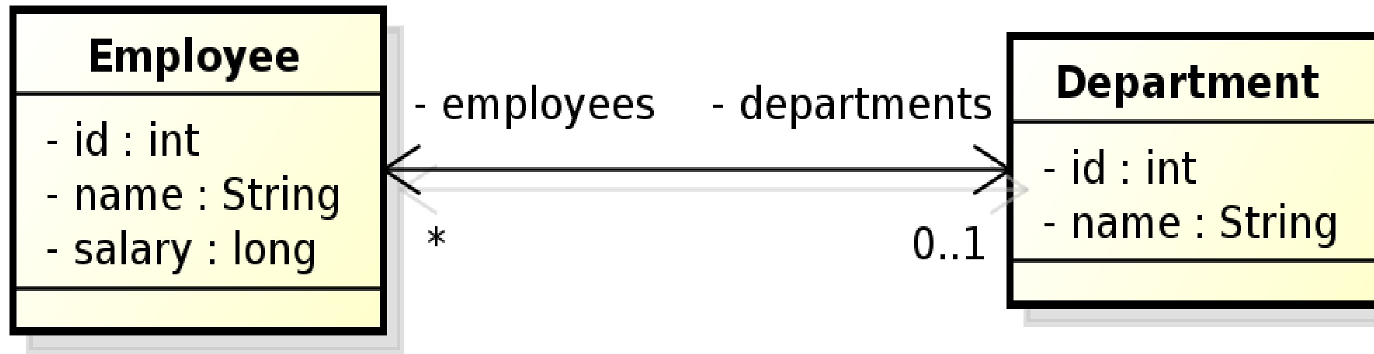
```java
@Embeddable
public class Birth {
   private String hasPlace;

   @Temporal(value=TemporalType.DATE)
   private Date hasDateOfBirth;
   // getters + setters
}
```

```
PERSON
===============================
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)
HASDATEOFBIRTH date
HASPLACE varchar(255)
```

Cascade Persist

```
@Entity
public class Employee {
    // …
    @ManyToOne(cascade=cascadeType.PERSIST)
    Address address;
    // …
}


Employee emp = new Employee();
emp.setId(2);
emp.setName(“Rob“);
Address addr = new Address();
addr.setStreet(“164 Brown Deer Road“);
addr.steCity(“Milwaukee“);
addr.setState(“WI“);
emp.setAddress(addr);
em.persist(addr);
em.persist(emp);
```

Persisting bidirectional relationship




```
…
Department dept = em.find(Deprtment.class, 101);
Employee emp = new Employee();
emp.setId(2);
emp.setName(“Rob“);
emp.setSalary(25000);
dept.employees.add(emp);  // @ManyToOne(cascade=cascadeType.PERSIST)
em.persist(dept);

!!! emp.departments still doesn't contain dept !!!

em.refresh(dept);

!!! emp.departments does contain dept  now !!!
```

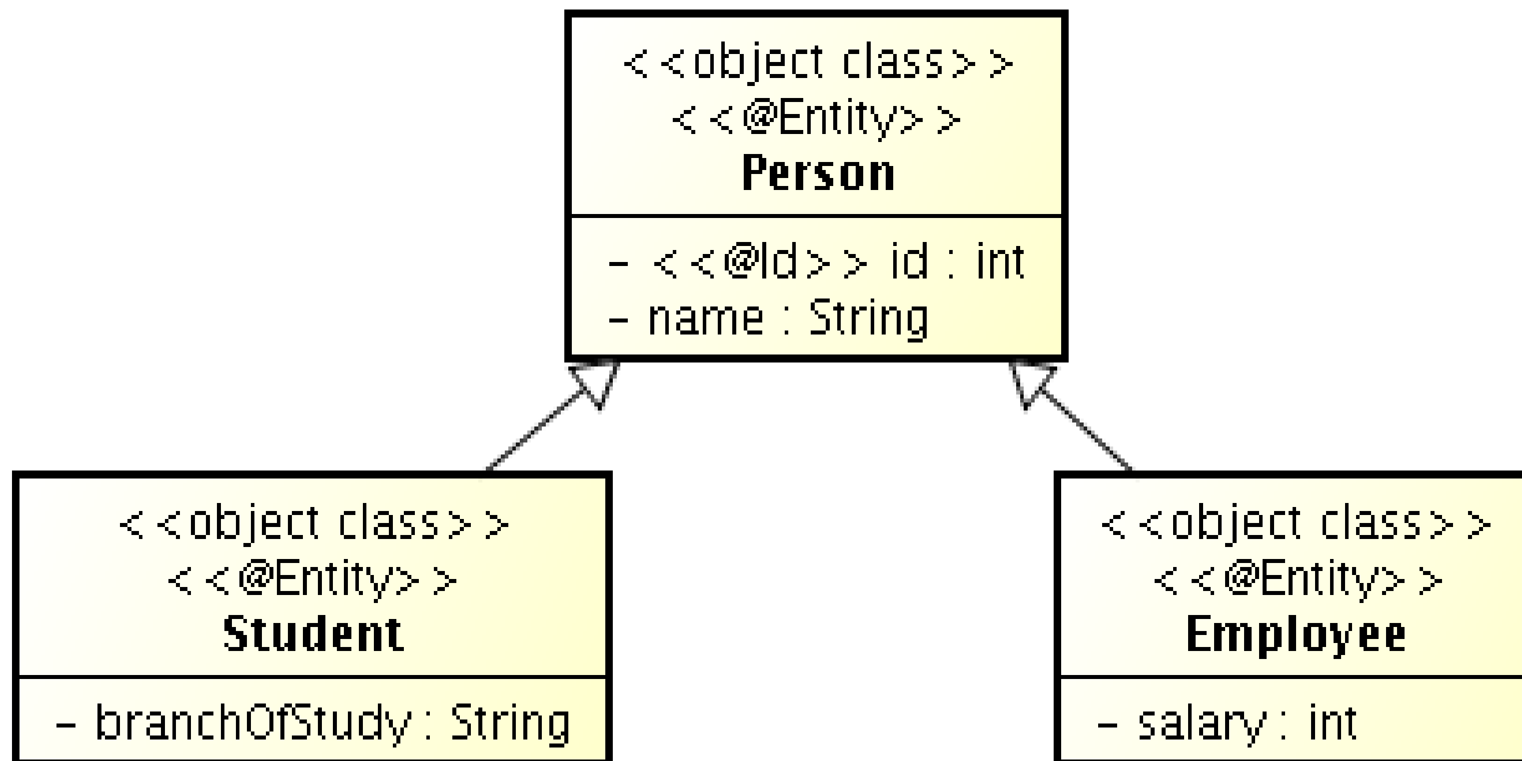
Cascade

List of operations supporting cascading:

- cascadeType.ALL
- cascadeType.DETACH
- cascadeType.MERGE
- cascadeType.PERSIST
- cascadeType.REFRESH
- cascadeType.REMOVE

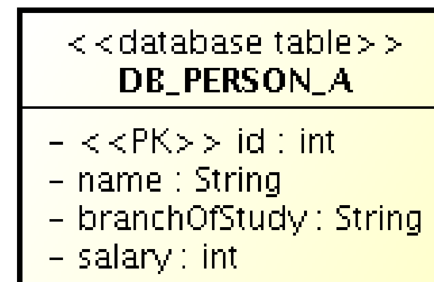
Inheritance

- How to map inheritance into RDBMS ?

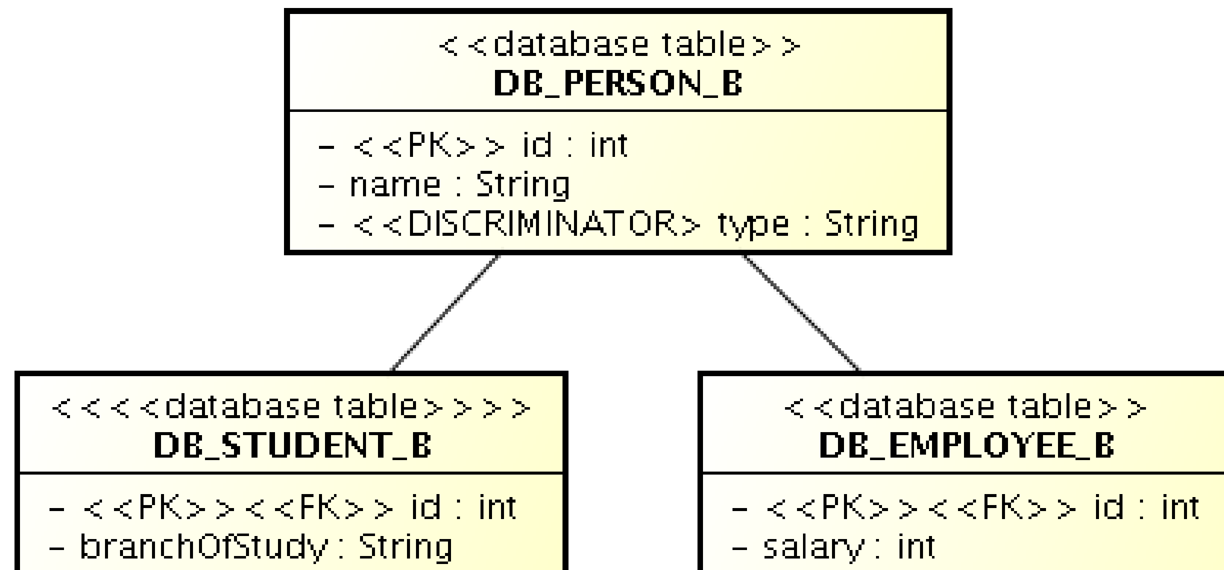


Strategies for inheritance mapping

- Single table

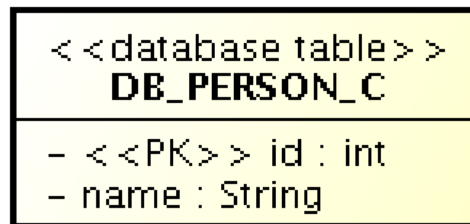


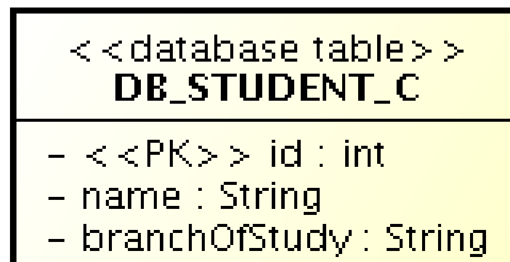
- Joined

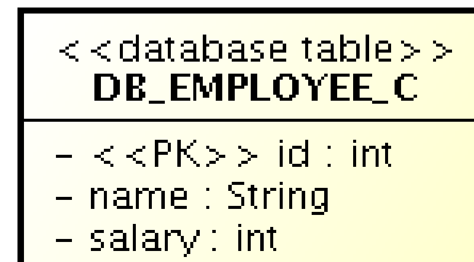


Strategies for inheritance mapping

- Table-per-concrete-class







Inheritance mapping
single-table strategy

```
@Entity
@Table(name=“DB_PERSON_C“)
@Inheritance  //same as @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminationColumn(name=“EMP_TYPE“)
public abstract class Person { …}

@Entity
@DiscriminatorValue(“Emp“)
Public class Employee extends Person {…}

@Entity
@DiscriminatorValue(“Stud“)
Public class Student extends Person {...}
```

Inheritance mapping
joined strategy

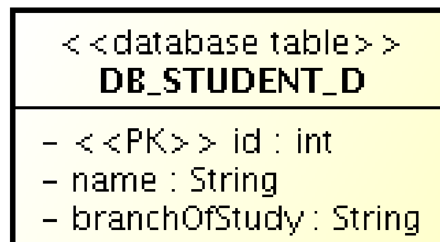
```
@Entity
@Table(name="DB_PERSON_C")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminationColumn(name="EMP_TYPE",
                      discriminatorType=discriminatorType.INTEGER)
public abstract class Person { …}

@Entity
@Table(name="DB_EMPLOYEE_C")
@DiscriminatorValue("1")
public class Employee extends Person {…}

@Entity
@Table(name="DB_STUDENT_C")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

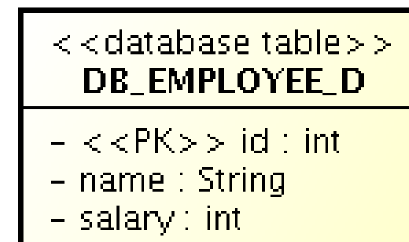
Inheritance mapping
table-per-concrete-class strategy

```
@Entity
@Table(name="DB_PERSON_C")
public abstract class Person { …}

@Entity
@Table(name="DB_EMPLOYEE_C")
@AttributeOverride(name="name", column=@Column(name="FULLNAME"))
@DiscriminatorValue("1")
public class Employee extends Person {…}

@Entity
@Table(name="DB_STUDENT_C")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

Strategies for inheritance mapping

- If Person is not an `@Entity`, but a `@MappedSuperClass`





- If `Person` is not an `@Entity`, neither `@MappedSuperClass`, the deploy fails as the `@Id` is in the `Person` (non-entity) class.

Collection Mapping

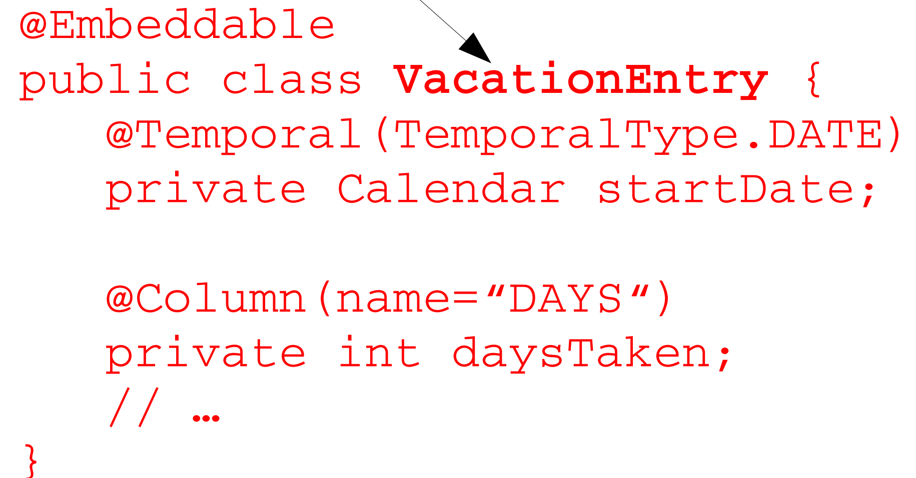
- Collection-valued relationship (above)
  - @OneToMany
  - @ManyToMany

- Element collections
  - @ElementCollection
  - Collections of Embeddable (new in JPA 2.0)
  - Collections of basic types (new in JPA 2.0)

- Specific types of Collections are supported
  - Lists
  - Maps

Collection Mapping

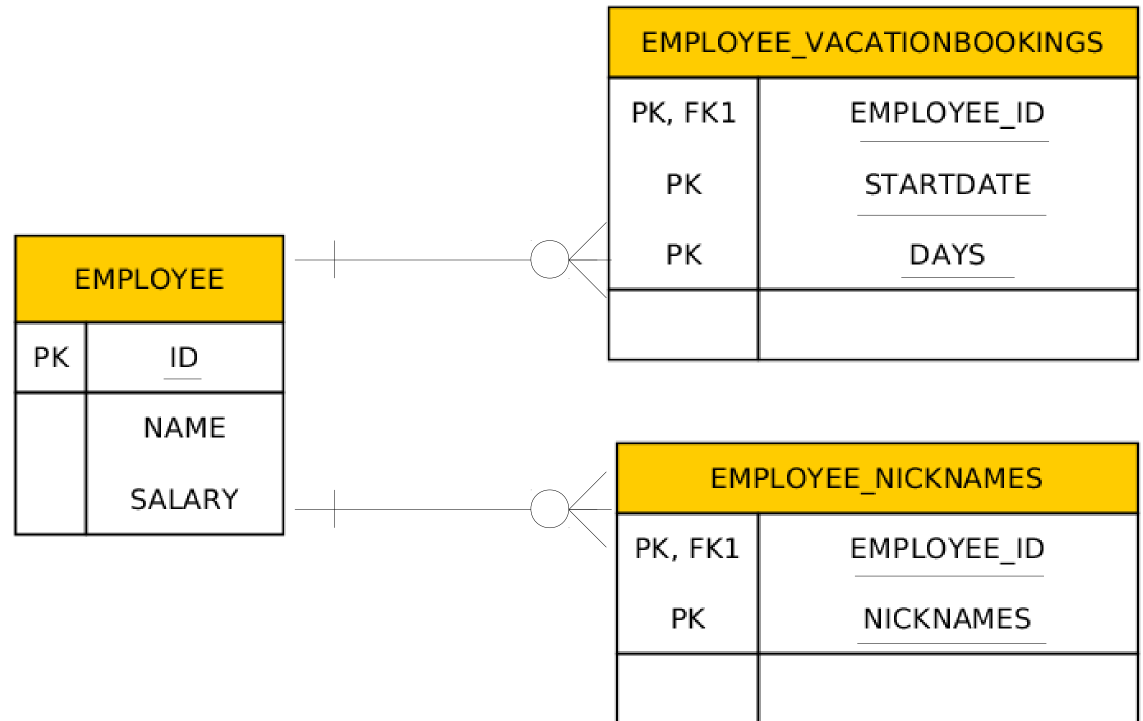
```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // …
    @ElementCollection(targetClass=VacationEntry.class);
    private Collection vacationBookings;

    @ElementCollection
    private Set<String> nickName;
    // …
}
```



```
@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;

    @Column(name=“DAYS“)
    private int daysTaken;
    // …
}
```

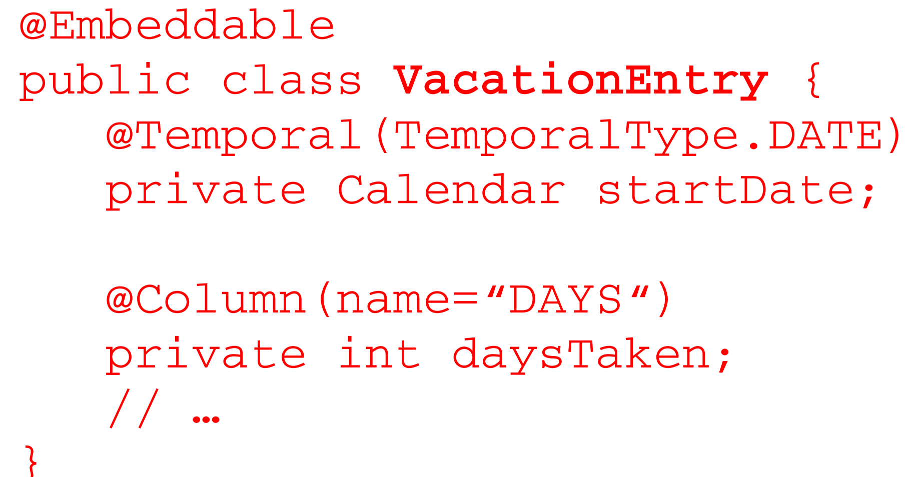
Collection Mapping

```
@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> nickName;
   // …
}
```



Collection Mapping

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // …
    @ElementCollection(targetClass=VacationEntry.class);
    @CollectionTable(
        name=“VACATION“,
        joinColumn=@JoinColumns(name=“EMP_ID“);
    @AttributeOverride(name=“daysTaken“, column=“DAYS_ABS“))
    private Collection vacationBookings;

    @ElementCollection
    @Column(name=“NICKNAME“)
    private Set<String> nickName;
    // …
}
```

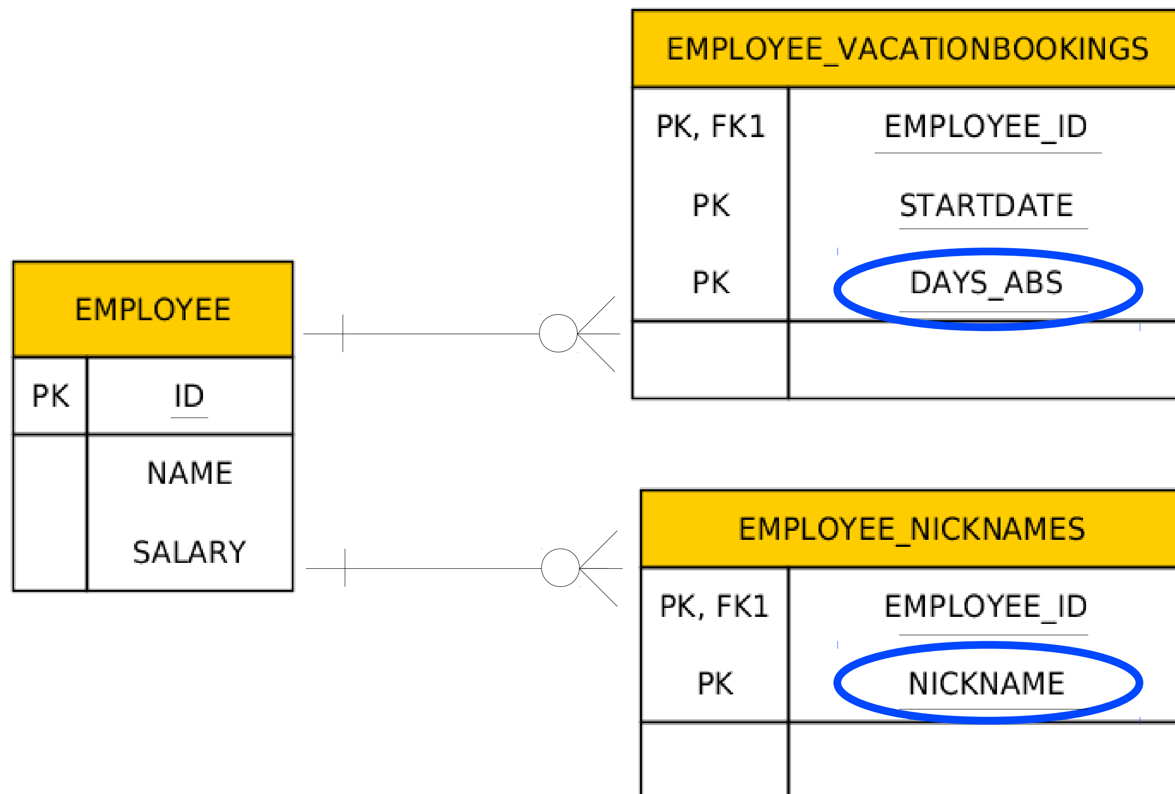


```
@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;

    @Column(name=“DAYS“)
    private int daysTaken;
    // …
}
```

Collection Mapping

```
@Entity
public class Employee {
   @Id private int id;
   private String name;
   private long salary;
   // …
   @ElementCollection(targetClass  = VacationEntry.class);
   @CollectionTable(
      name="VACATION",
      joinColumn=@JoinColumns(name="EMP_LO
   @AttributeOverride(name="daysTaken", column="DAYS_ABS"))
   private Collection vacationBookings;

   @ElementCollection
    @Column(name="NICKNAME")
    private Set<String> nickName;
   // …
}
```

```
@Embeddable
public class VacationEntry {
   @Temporal(TemporalType.DATE)
   private Calendar startDate;

   @Column(name="DAYS")
   private int daysTaken;
   // …
}
```



Collection Mapping

Interfaces:

- Collection
- Set
- List
- Map

may be used for mapping purposes.

An instance of an appropriate implementation class (HashSet, ArrayList, etc.) will be used to implement the respective property initially (the entity will be unmanaged).

As soon as such an Entity becomes managed (by calling em.persist(...)), we can expect to get an instance of the respective interface, not an instance of that particular implementation class.

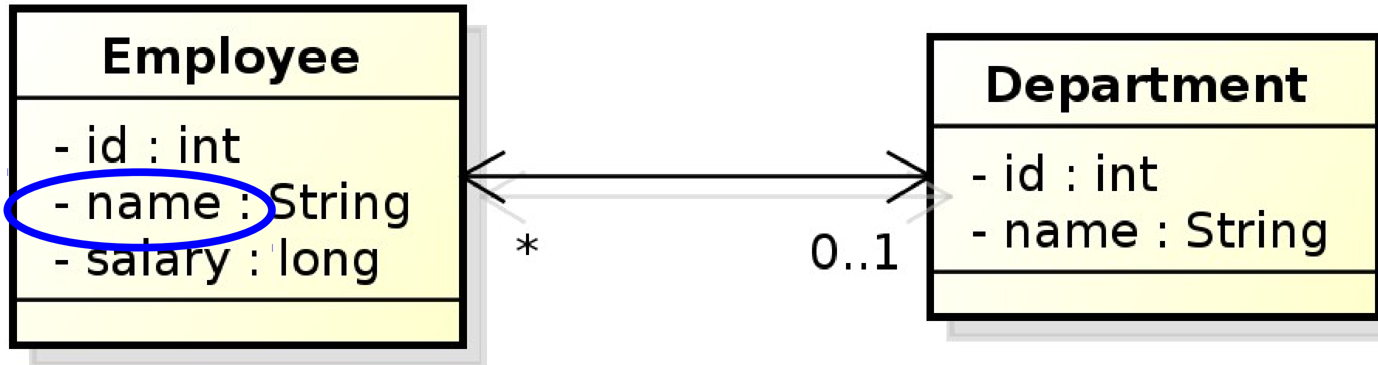
When we get it back (em.find(..)) to the persistence context. The reason is that the JPA provider may replace the initial concrete instance with an alternate instance of the respective interface (Collection, Set, List, Map).

Collection Mapping – ordered List

- Ordering by Entity or Element Attribute
  ordering according to the state that exists in each entity or element in the List

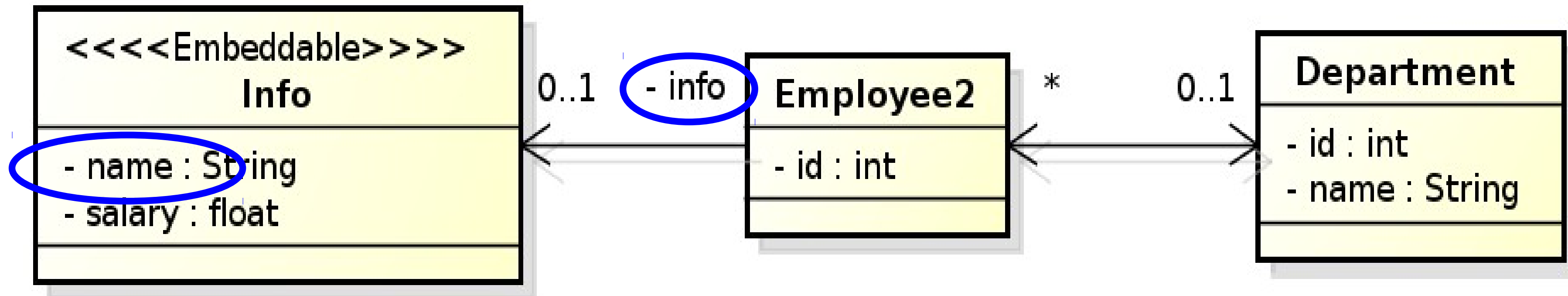
- Persistently ordered lists
  the ordering is persisted by means of an additional database column(s)
  typical example – ordering = the order in which the entities were persisted

Collection Mapping – ordered List
(Ordering by Entity or Element Attribute)



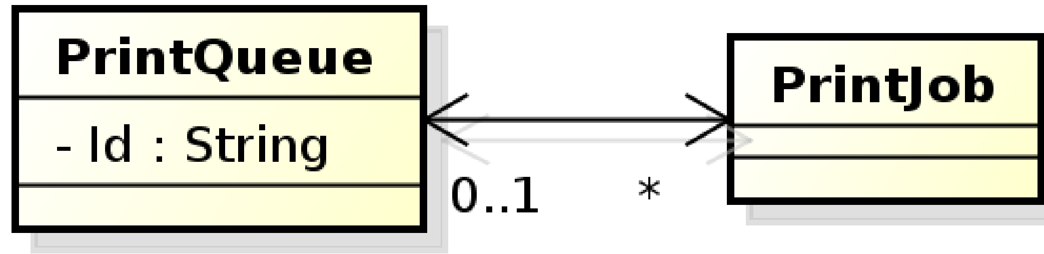

```
@Entity
public class Department {
    // …
    @OneToMany(mappedBy="department")
    @OrderBy("name ASC")
    private List<Employee> employees;
    // …
}
```

Collection Mapping – ordered List
(Ordering by Entity or Element Attribute)



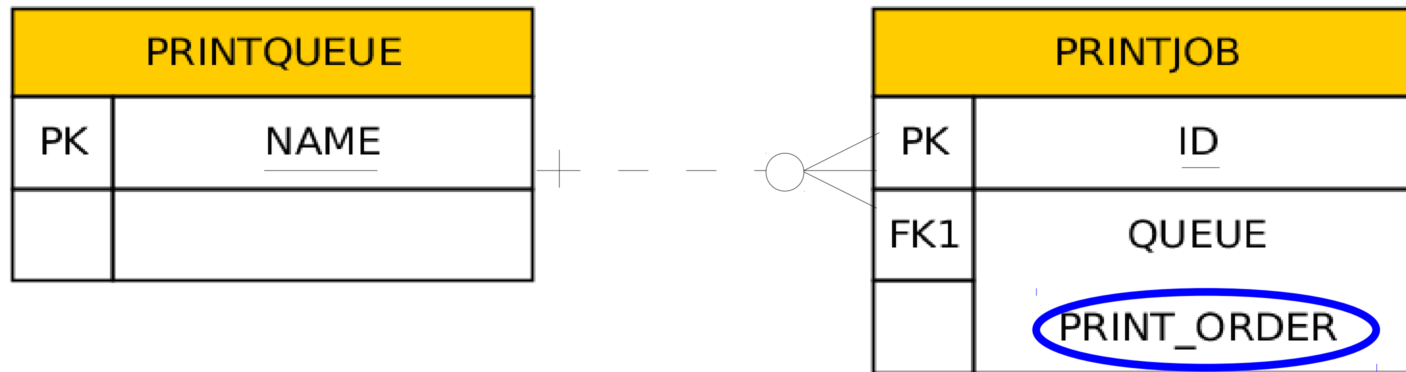

```
@Entity
public class Department {
    // …
    @OneToMany(mappedBy="department")
    @OrderBy("info.name ASC")
    private List<Employee2> employees;
    // …
}
```

Collection Mapping – ordered List
(Persistently ordered lists)




```
@Entity
public class PrintQueue {
    @Id private String name;
    // …
    @OneToMany(mappedBy="queue")
    @OrderColumn(name="PRINT_ORDER")
    private List<PrintJob> jobs;
    // …
}
```

Collection Mapping – ordered List
(Persistently ordered lists)




```
@Entity
public class PrintQueue {
    @Id private String name;
    // …
    @OneToMany(mappedBy="queue")
    @OrderColumn(name="PRINT_ORDER")
    private List<PrintJob> jobs;
    // …
}
```

This annotation need not be necessarily on the owning side

Collection Mapping – Maps

Map is an object that maps keys to values.
A map cannot contain duplicate keys;
each key can map to at most one value.

Keys:

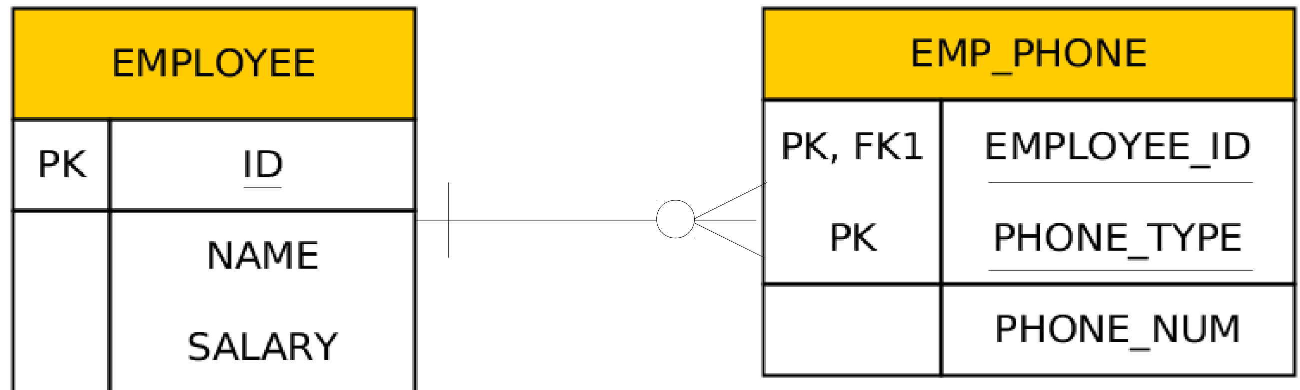
- Basic types (stored directly in the table being referred to)
  - Target entity table
  - Join table
  - Collection table
- Embeddable types (- “ -)
- Entities (only foreign key is stored in the table)

Values:

- Values are entities => Map must be mapped as a one-to-many or many-to-many relationship
- Values are basic types ot embeddable types => Map is mapped as an element collection

Collection Mapping – Maps
(keying by basic type – key is String)

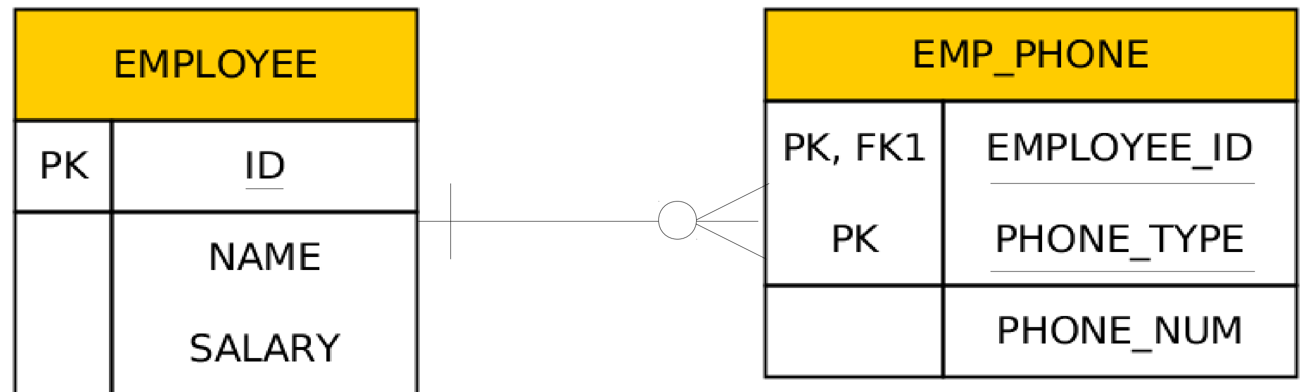
```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<String, String> phoneNumbers;
    // …
}
```



Collection Mapping – Maps
(keying by basic type – key is an enumeration)

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyEnumerated(EnumType.String)
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<PhoneType, String> phoneNumbers;
    // …
}
```

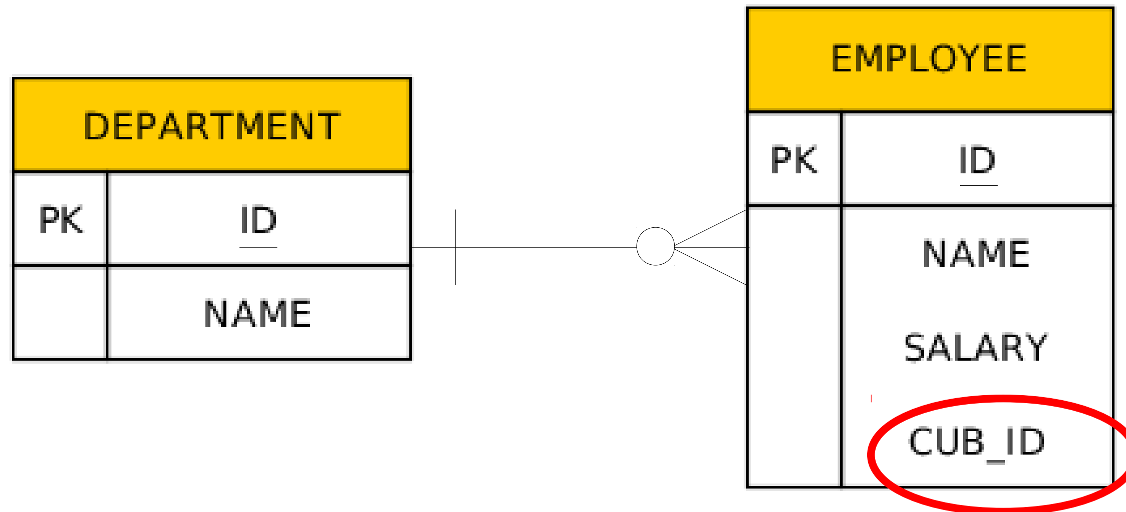
```
Public enum PhoneType {
        Home,
        Mobile,
        Work
}
```



Collection Mapping – Maps

(keying by basic type – 1:N relationship using a Map with String key)

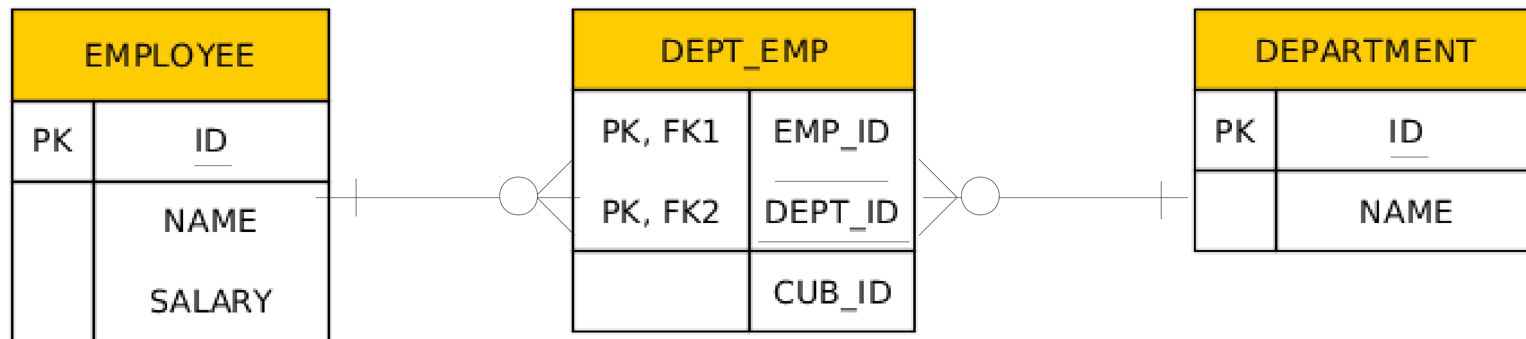
```
@Entity
public class Department {
   @Id private int id;
   private String name;

   @OneToMany(mappedBy="department")
   @MapKeyColumn(name="CUB_ID")
   private Map<String, Employee> employeesByCubicle;
   // …
}
```



Collection Mapping – Maps

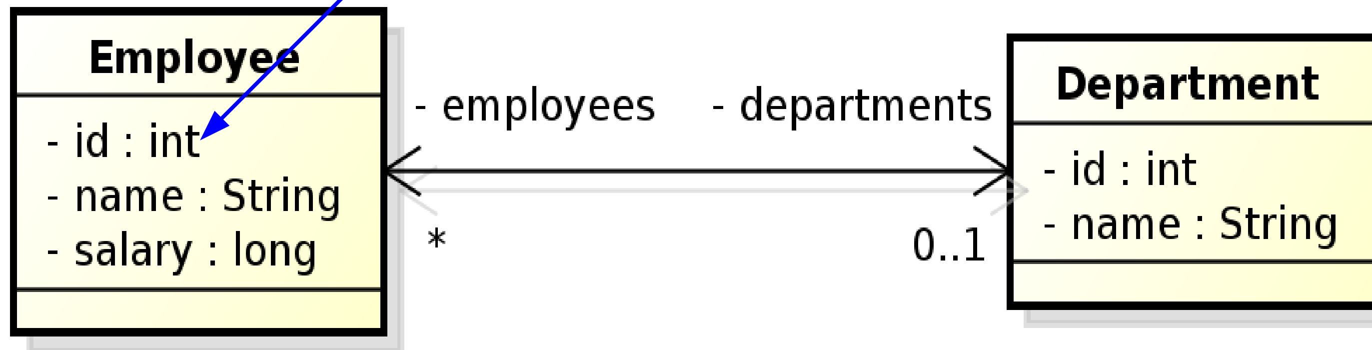
(keying by basic type – N:M relationship using a Map with String key)

```
@Entity
public class Department {
    @Id private int id;
    private String name;

    @ManyToMany
    @JoinTable(name="DEPT_EMP",
        joinColumns=@JoinColumn(name="DEPT_ID"),
        inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // …
}
```



Collection Mapping – Maps
(keying by entity attribute)

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="id")
    private Map<Integer, Employee> employees;
    // …
}
```



Collection Mapping – Maps
(keying by embeddable type)

```
@Entity
Public class Employee {
    @Id private int id;
    @Column(name="F_NAME");
    private String firstName;
    @Column(name="L_NAME");
    private String lastName;
    private long salary;
    //...
}
```

Sharing columns => insertable=false and updatable = false

```
@Embeddable
public class EmployeeName {
    @Column(name="F_NAME", insertable=false,
                           updateble=false)
    private String first_Name;
    @Column(name="L_NAME", insertable=false,
                           updateble=false)
    private String last_Name;
    // …
}
```

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="id")
    private Map<EmployeeName, Employee> employees;
    // …
}
```

Collection Mapping – Maps

(keying by embeddable type)

```
@Entity
Public class Employee {
    @Id private int id;

    @Embedded
    private EmployeeName name;
    private long salary;
    //...
}
```

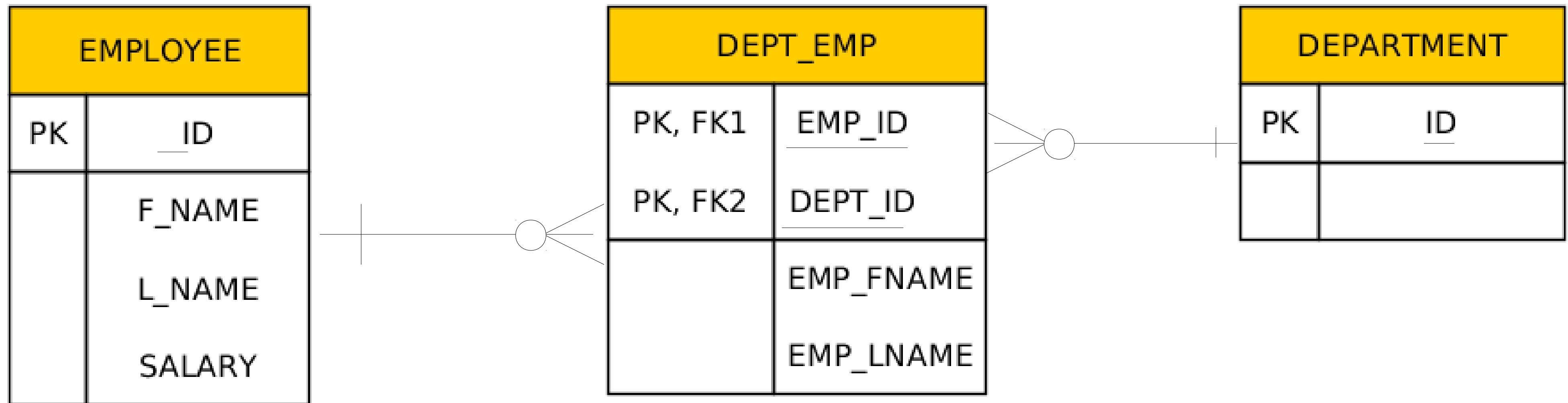
Columns are not shared

```
@Embeddable
Public class EmployeeName {
    @Column(name="F_NAME", insertable=false,
                                  updateble=false)
    Private String first_Name;
    @Column(name="L_NAME", insertable=false,
                                  updateble=false)
    Private String last_Name;
    // …
}
```

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="id")
    private Map<EmployeeName, Employee> employees;
    // …
}
```

Collection Mapping – Maps
(keying by embeddable type)

```
@Entity
public class Department {
    @Id private int id;
    @ManyToMany
    @JoinTable(name="DEPT_EMP",
        joinColumns=@JoinColumn(name="DEPT_ID"),
        inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @AttributeOverrides({
        @AttributeOverride(
            name="first_Name",
            column=@Column(name="EMP_FNAME")),
```



Collection Mapping – Maps
(keying by embeddable type)

```
@Entity
public class Department {
    @Id private int id;
    @ManyToMany
    @JoinTable(name="DEPT_EMP",
        joinColumns=@JoinColumn(name="DEPT_ID"),
        inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @AttributeOverrides({
        @AttributeOverride(
            name="first_Name",
            column=@Column(name="EMP_FNAME")),
        @AttributeOverride(
            name="last_Name",
            column=@Column(name="EMP_LNAME"))
    })
    private Map<EmployeeName, Employee> employees;
    // …
}
```

Collection Mapping – Maps
(keying by embeddable type)

We have to distinguish, if we are overriding embeddable attributes of the key or the value.

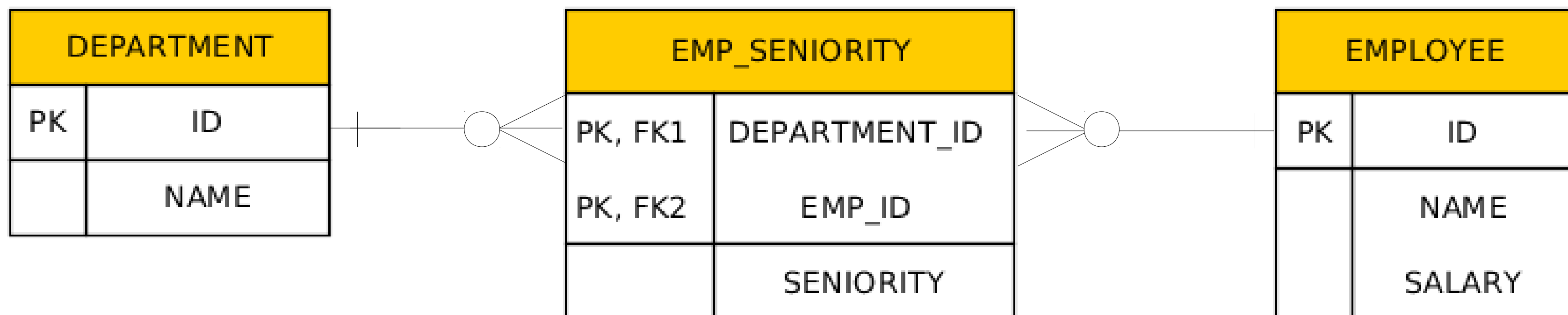
```
@Entity
public class Department {
    @Id private int id;
    @AttributeOverrides({
        @AttributeOverride(name="key.first_Name",
                               column=@Column(name="EMP_FNAME")),
        @AttributeOverride(name="key.last_Name",
                               column=@Column(name="EMP_LNAME"))
    })
    private Map<EmployeeName, EmployeeInfo> employees;
    // …
}
```

The embeddable attributes will be stored in the collection table (rather than in a join table As it was on the previous slide).

Collection Mapping – Maps
(keying by entity)

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    // …
    @ElementCollection
    @CollectionTable(name="EMP_SENIORITY")
    @MapKeyJoinColumn(name="EMP_ID")
    @Column(name="SENIORITY")
    private Map<Employee, Integer> employees;
    // …
}
```

Collection table



Compound primary keys

Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

No setters. Once created, can not be changed.

```
public class EmployeeId
                 implements Serializable {
   private String country;
   private int id;

   public EmployeeId() {}
   public EmployeeId(String country,
                     int id) {
      this.country = country;
      this.id = id;
   }

   public String getCountry() {...};
   public int getId() {...}

   public boolean equals(Object o) {...}

   public int hashCode() {
      Return country.hashCode() + id;
   }
```

```
@Entity
@IdClass(EmployeeId.class)
public class Emlpoyee {
   @Id private String country;
   @Id
   @Column(name="EMP_ID")
   private int id;
   private String name;
   private long salary;
   // …
}
```

```
EmployeeId id = new Employee(country, id);
Employee emp = em.find(Employee.class, id);
```

Compound primary keys
Embedded Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

```
@Embeddable
public class EmployeeId
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
                      int id) {
        this.country = country;
        this.id = id;
    }
    // ...
}
```

```
@Entity
public class Emlpoyee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;
    // …
    public String getCountry() {return id.getCountry();}
    Public int getId() {return id.getId();}
    // …
}
```

Compound primary keys
Embedded Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

```
@Embeddable
public class EmployeeId
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
                      int id) {
        this.country = country;
        this.id = id;
    }
```

Referencing an embedded IdClass in a query:

```
em.createQuery("SELECT e FROM Employee e " +
               "WHERE e.id.country = ?1 AND e.id.id = @2")
      .setParameter(1, country)
      .setParameter(2, id)
      .getSingleResult();
```

Shared Primary Key

Bidirectional one-to-one relationship between Employee and EmployeeHistory

```
@Entity
public class EmployeeHistory
    // ...
    @Id
    @OneToOne
    @JoinColumn(name="EMP_ID")
    private Employee employee;
    // ...
}
```

The primary key type of EmployeeHistory is the same as primary key of Employee.

- If <pk> of Emloyee is integer, <pk> of EmployeeHistory will be also integer.
- If Employee has a compound <pk>, either with an id class or an embedded id class, then EmployeeHistory will share the same id class and should also be annotated
- @IdClass.

The rule is that a primary key attribute corresponds to a relationship attribute. However, the relationship attribute is missing in this case (the id class is shared between both parent and dependententities). Hence, this is an exception from the above mentioned rule.

Shared Primary Key

Bidirectional one-to-one relationship between Employee and EmployeeHistory

```
@Entity
public class EmployeeHistory
    // ...
    @Id
    int empId;

    @MapsId
    @OneToOne
    @JoinColumn(name="EMP_ID")
    private Employee employee;
    // ...
}
```

On the provious slide, the relationship attribute was missing.

In this case, the EmployeeHistory class contains both a primary key attribute as well as the relation-ship attribute. Both attributes are mapped to the same foreign key column in the table.

@MapsId annotates the relationship attribute to indicate that it is mapping the id attribute as well (**read-only mapping**!). Updates/inserts to the foreign key column will only occur through the relationship attribute.

=> YOU MUST ALWAYS SET THE PARENT RELATIONSHIPS BEFORE TRYING TO PERSIST A DEPENDENT ENTITY.

Read-only mappings

The constrains are checked on commit!
Hence, the constrained properties can be
Modified in memory.

```
@Entity
public class Employee
    @Id
    @Column(insertable=false)
    private int id;

    @Column(insertable=false, updatable=false)
    private String name;

    @Column(insertable=false, updatable=false)
    private long salary;

    @ManyToOne
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

Optionality

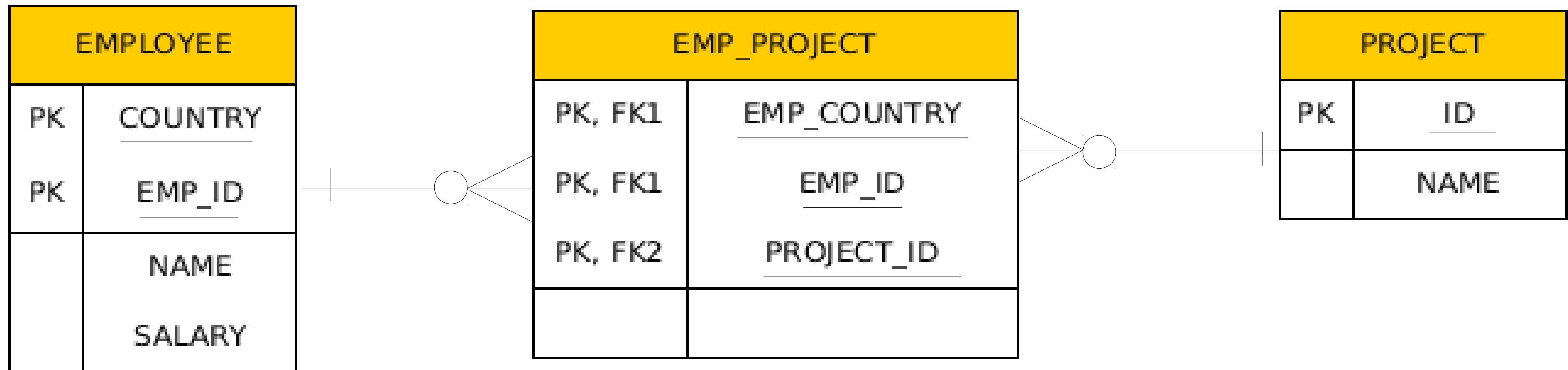
```
@Entity
public class Employee
    // ...

    @ManyToOne(optional=false)
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

Compound Join Columns

	EMPLOYEE
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY
FK1	MGR_COUNTRY
FK1	MGR_ID

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column name="EMP_ID")
    private int id;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MGR_COUNTRY",
                    referencedColumnName="COUNTRY"),
        @JoinColumn(name="MGR_ID",
                    referencedColumnName="EMP_ID")
    })
    private Employee manager;

    @OneToMany(mappedBy="manager")
    private Collection<Employee> directs;
    // …
}
```

Compound Join Columns




```
@Entity
@IdClass(EmployeeId.class)
public class Employee
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    @ManyToMany
    @JoinTable(
        name="EMP_PROJECT",
        joinColumns={
            @JoinColumn(name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
            @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")},
        inverseJoinColumns=@JoinColumn(name="PROJECT_ID"))
    private Collection<Project> projects;
}
```

Access types – Field access

```
@Entity
public class Employee {
   @Id
   private int id;
   …
   public int getId() {return id;}
   public void set Id(int id) {this.id=id;}
   …
}
```

The provider will get and set the fields of the entity using reflection (not using getters and setters).

Access types – Property access

```
@Entity
public class Employee {
   private int id;
   …
   @Id
   public int getId() {return id;}
   public void set Id(int id) {this.id=id;}
   …
}
```

Annotation is placed in front of getter.
(Annotation in front of setter abandoned)

The provider will get and set the fields of the entity by invoking getters and setters.

Access types – Mixed access

- Field access with property access combined within the same entity hierarchy (or even withinthe same entity).

- @Access – defines the default access mode
  (may be overriden for the entity subclass)

- An example on the next slide

Access types – Mixed access

```
@Entity @Access(AccessType.FIELD)
public class Employee {
    public static final String LOCAL_AREA_CODE = “613“;
    @Id private int id;
    @Transient private String phoneNum;
    …
    public int getId() {return Id};
    public void setId(int id) {this.id = id;}

    public String getPhoneNumber() {return phoneNum;}
    public void setPhoneNumber(Strung num) {this.phoneNum=num;}

    @Access(AccessType.PROPERTY) @Column(name=“PHONE“)
    protected String getPhoneNumberForDb() {
        if (phoneNum.length()==10) return phoneNum;
        else return LOCAL_AREA_CODE + phoneNum;
    }
    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3);
        else phoneNum = num;
}
```

Queries

- JPQL (Java Persistence Query Language)
- Native queries (SQL)
- Criteria API
  - queries represented as Java Objects (not strings)
  - using Metamodel API to model the persistence unit.

JPQL

JPQL very similar to SQL (especially in JPA 2.0)

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND p.type = 'CELL'
```

Conditions do not stick on values of database columns, but on entities and their properties.

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

JPQL – query parameters

- positional

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND e.salary > ?2
```

- named

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND salary > :base
```

JPQL – defining a query dynamically

```
@Stateless
public class QueryServiceBean implements QueruService {
   @PersistenceContext(unitName=“DynamicQueries“)
   EntityManager em;

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

JPQL – using parameters

```
String QUERY = "SELECT e.salary FROM Employee e " +
                     "WHERE e.department.name = :deptName " +
                     "AND e.name = :empName";

public long queryEmpSalary(String deptName, String empName) {
   return em.createQuery(QUERY, Long.class)
               .setParameter("deptName", deptName)
               .setParameter("empName", empName)
               .getSingleResult();
}
```

JPQL – named queries

```
@NamedQuery(name="Employee.findByName",
            query="SELECT e FROM Employee e " +
                  "WHERE e.name = :name")
```

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

JPQL – named queries

```
@NamedQuery(name="Employee.findByDept",
            query="SELECT e FROM Employee e " +
                  "WHERE e.department = ?1")
```

```
public void printEmployeesForDepartment(String dept)  {
   List<Employee> result =
       em.createNamedQuery("Employee.findByDept",
                            Employee.class)
            .setParameter(1, dept)
            .getResultList();
   int count = 0;
   for (Employee e: result) {
      System.out.println(++count + ":" + e.getName);
   }
}
```

JPQL – pagination

```
private long pageSize    = 800;
private long currentPage = 0;

public List getCurrentResults()  {
   return em.createNamedQuery("Employee.findByDept",
                              Employee.class)
            .setFirstResult(currentPage * pageSize)
            .setMaxResults(pageSize)
            .getResultList();
}

public void next() {
   currentPage++;
}
```

JPQL – bulk updates

Modifications of entities not only by em.persist() or em.remove();

```
em.createQuery("UPDATE Employee e SET e.manager = ?1 " +
                      "WHERE e.department = ?2)
                 .setParameter(1, manager)
                 .setParameter(2, dept)
                 .executeUpdate();
```

```
em.createQuery("DELETE FROM Project p „ +
               "WHERE p.employees IS EMPTY")
                 .executeUpdate();
```

If REMOVE cascade option is set fro a relationship, cascading remove occurs.

Native SQL update and delete operations should not be applied to tables mapped by an entity (transaction, cascading).

Native (SQL) queries

```
@NamedNativeQuery(
   name="getStructureReportingTo",
   query = "SELECT emp_id, name, salary, manager_id,"+
               "dept_id, address_id " +
               "FROM emp ",
   resultClass = Employee.class
)
```

Mapping is straightforward

Native (SQL) queries

```
@NamedNativeQuery(
   name="getEmployeeAddress",
   query = "SELECT emp_id, name, salary, manager_id,"+
               "dept_id, address_id, id, street, city, " +
               "state, zip " +
               "FROM emp JOIN address "
                          "ON emp.address_id = address.id)"
)
```

Mapping less straightforward

```
@SqlResultSetMapping(
   name="EmployeeWithAddress",
   entities={@EntityResult(entityClass=Employee.class),
             @EntityResult(entityClass=Address.class)}
```

Native (SQL) queries

```
Query q = em.createNativeQuery(
        "SELECT o.id AS order_id, " +
            "o.quantity AS order_quantity, " +
            "o.item AS order_item, " +
            "i.name AS item_name, " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderResults");

    @SqlResultSetMapping(name="OrderResults",
        entities={
            @EntityResult(entityClass=com.acme.Order.class,
        fields={
                @FieldResult(name="id", column="order_id"),
                @FieldResult(name="quantity",
                             column="order_quantity"),
                @FieldResult(name="item",
                             column="order_item")})},
        columns={
            @ColumnResult(name="item_name")}
    )
```

References

- Křemen, Kouba: Persistence, ORM, 2013, https://cw.felk.cvut.cz/wiki/_media/courses/a7b39wpa/jpa20.pdf
- JSR 221: JDBC 4.0, Sun Microsystems. 2006, http://download.oracle.com/otndocs/jcp/jdbc-4.0-fr-oth-JSpec
- JSR 317: Java Persistence API, Version 2.0, Sun Microsystems. 2009, http://download.oracle.com/otn-pub/jcp/persistence-2.0-fr-oth-JSpec/persistence-2_0-final-spec.pdf
- Goncalves A.: Beginning Java EE 6 Platform with Glassfish 3, 2009, http://www.apress.com/9781430219545