Advanced Topics in Accessing Persistent Data

Petr Křemen

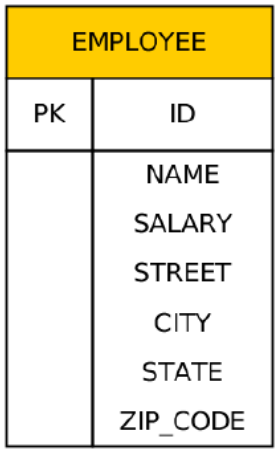
petr.kremen@fel.cvut.cz

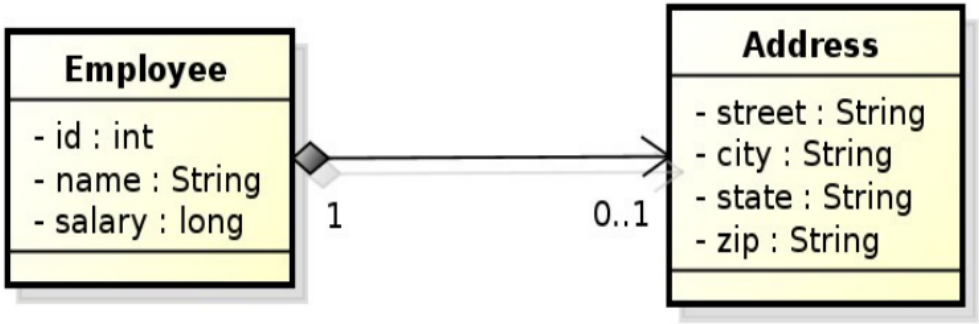
Winter Term 2018

Contents

Advanced JPA

Embedded Objects

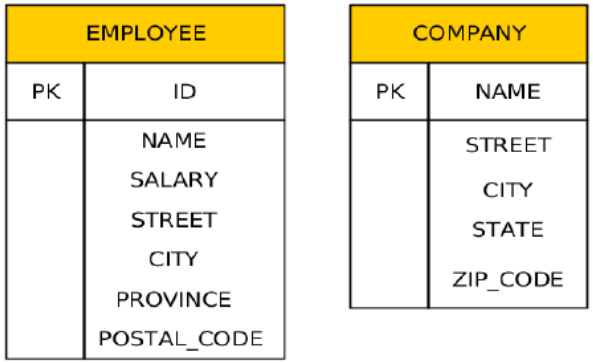


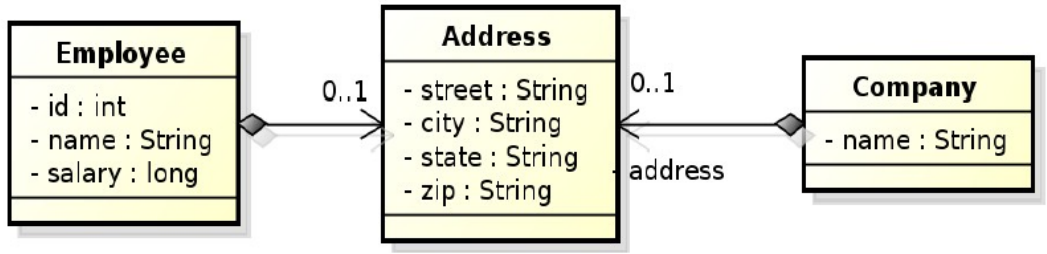



```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
   private String street;
   private String city;
   private String state;
   @Column(name="ZIP_CODE")
   private String zip;
}
```

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

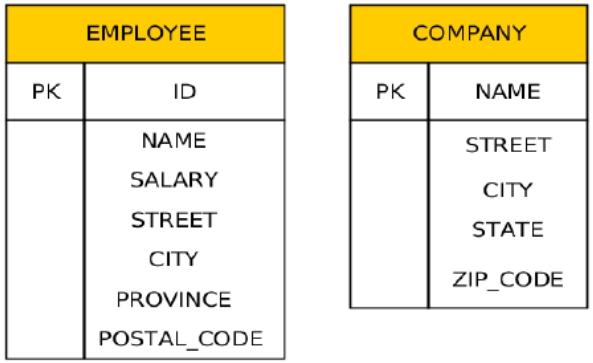
Embedded Objects

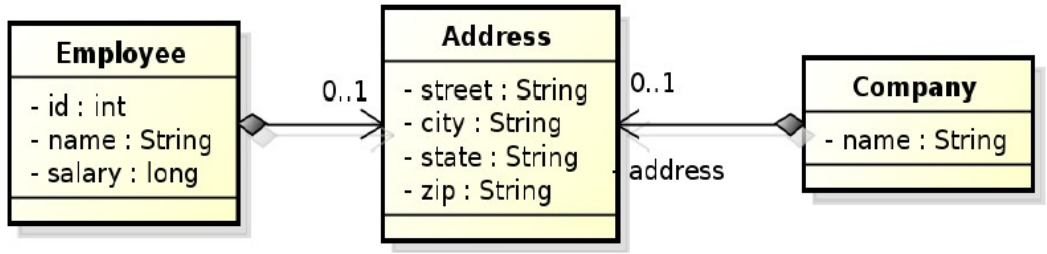





```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

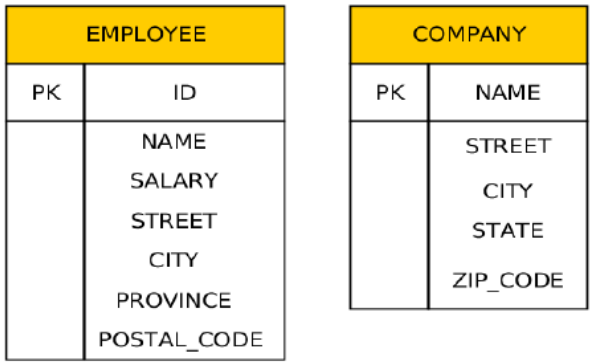
Embedded Objects

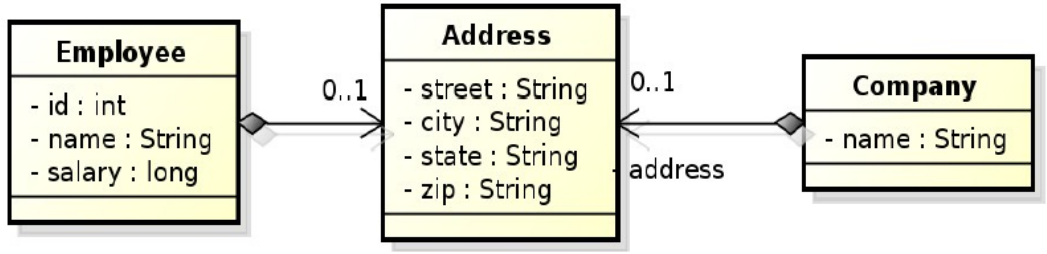





```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
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

Embedded Objects






```
@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*

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

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

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

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

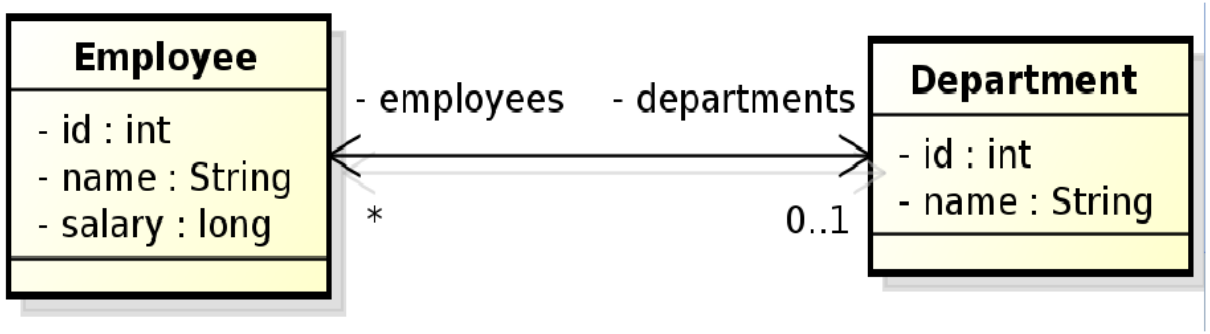
```
PERSON
==============================
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)
HASDATEOFBIRTH date
HASPLACE varchar(255)
```

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

Collection Mapping

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;
   // …
}
```

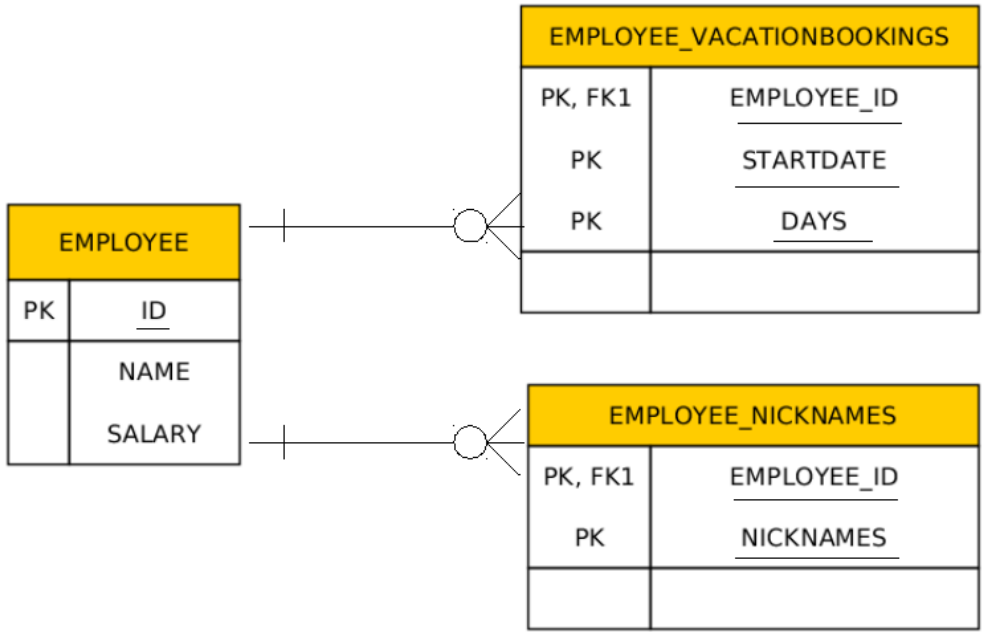
```
@Embeddable
public class VacationEntry {
   @Temporal(TemporalType.DATE)
   private Calendar startDate;

   @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;
   // …
}
```

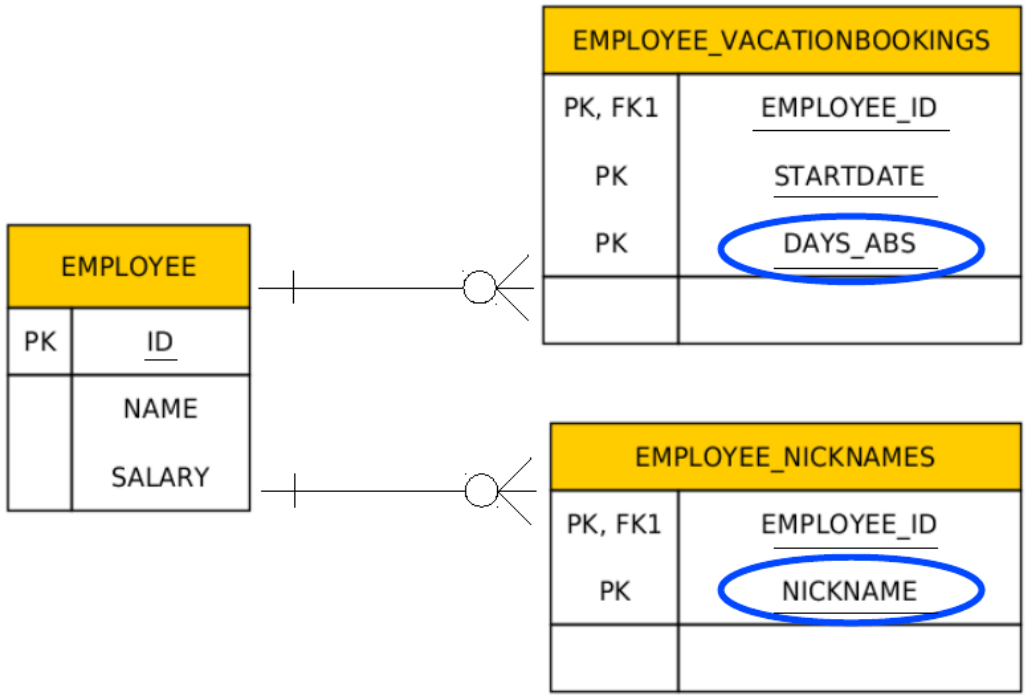
Collection Mapping

```
@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", colum="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.

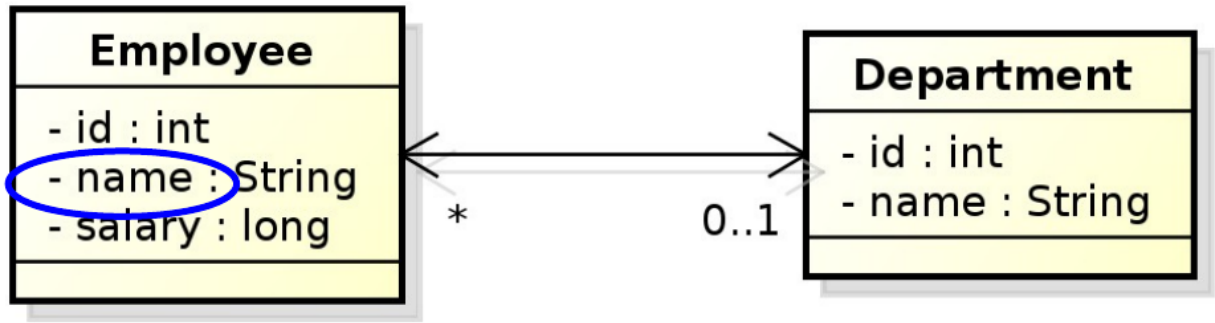
When we get it back (em.find(..)) to the persistence context. The reason is that the JPA provider may replace the initial concrete instance with an alternate instance of the respective interface (Collection, Set, List, Map).

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

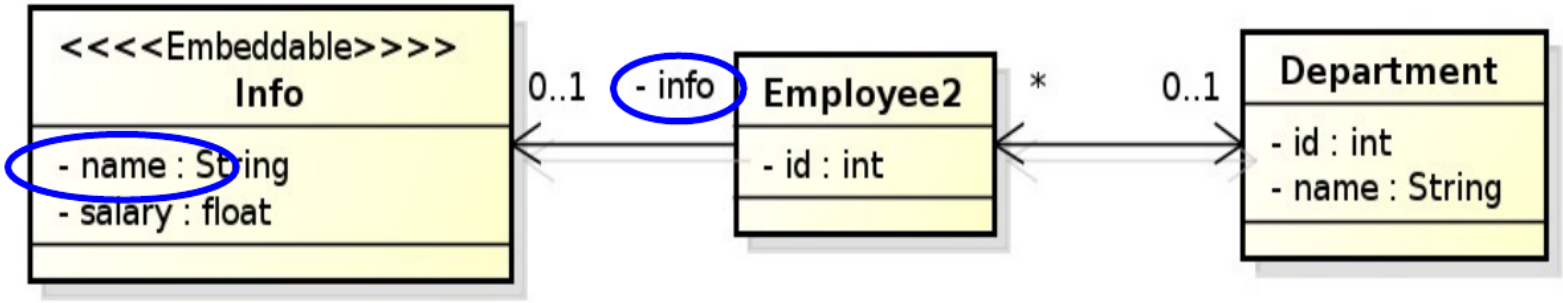
Collection Mapping - Ordered List




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

Figure: Ordering by Entity or Element Attribute

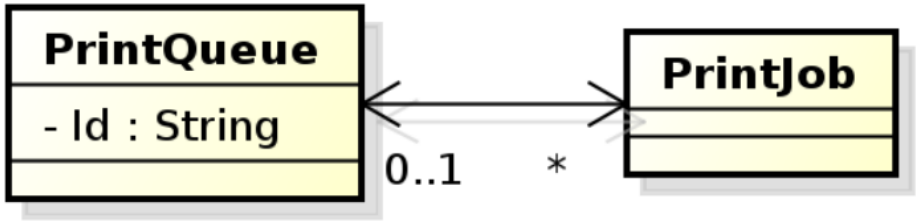
Collection Mapping - Ordered List




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

Figure: Ordering by Entity or Element Attribute

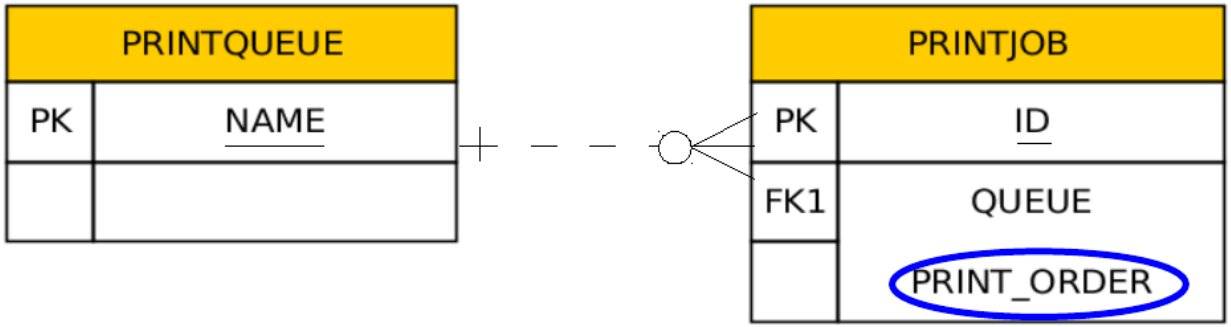
Collection Mapping - Ordered List




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

Figure: Persistently ordered lists

Collection Mapping - Ordered List




```
@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

Figure: Persistently ordered lists

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

```
@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;
    // …
}
```

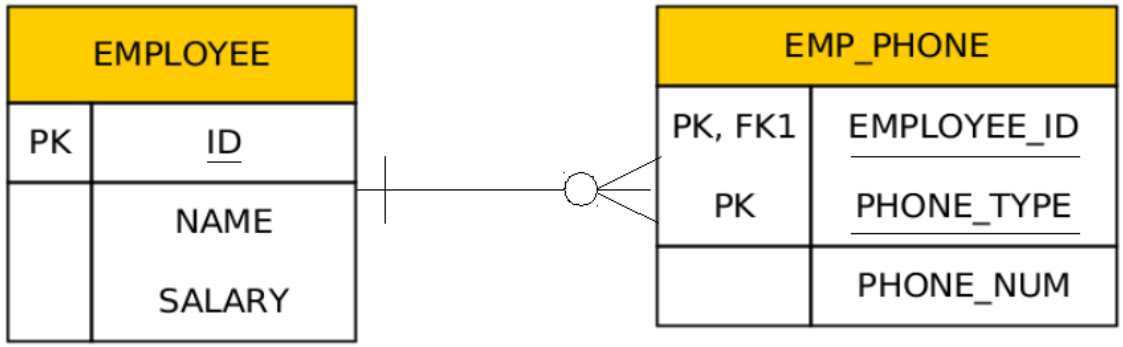


Figure: Keying by basic type - key is String

Collection Mapping - Maps

```
@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
}
```

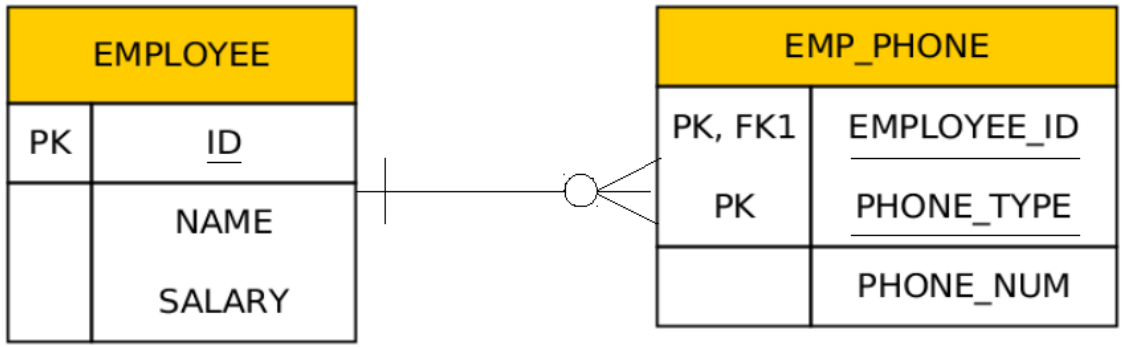


Figure: Keying by basic type - key is an enumeration

Collection Mapping - Maps

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

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

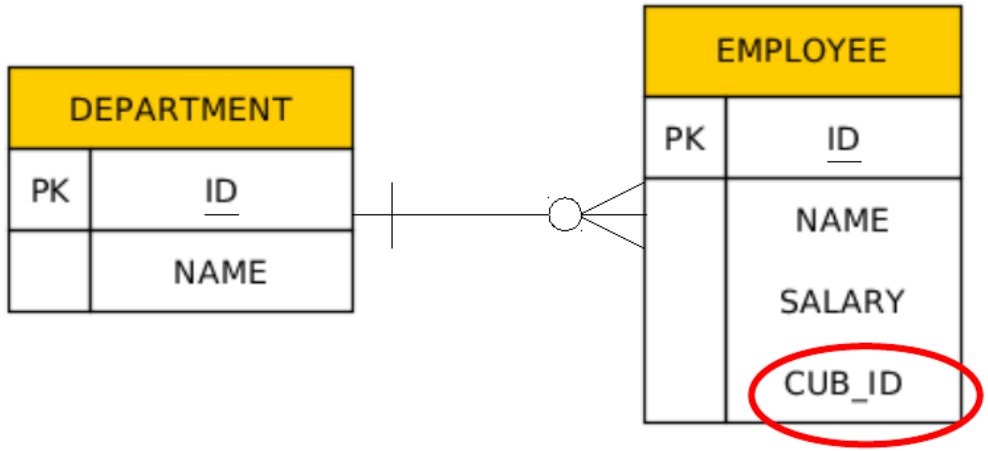


Figure: Keying by basic type - 1:N relationship using a Map with String key

Collection Mapping - Maps

```
@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;
    // …
}
```

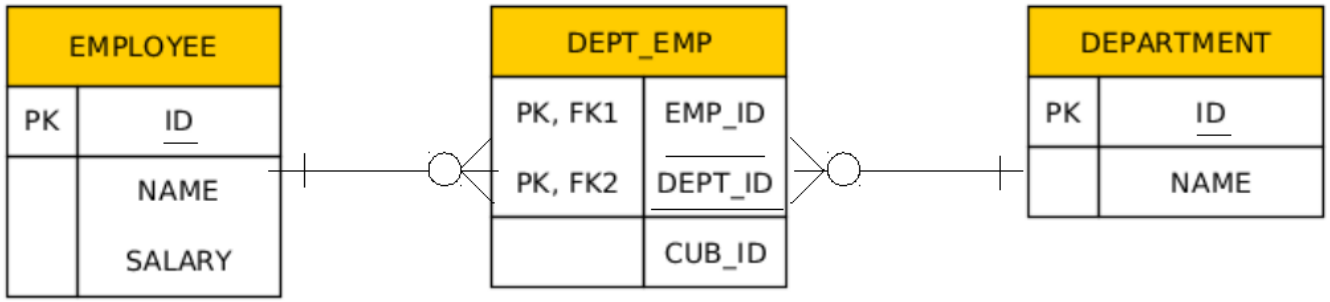


Figure: Keying by basic type - N:M relationship using a Map with String key

Collection Mapping - Maps

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

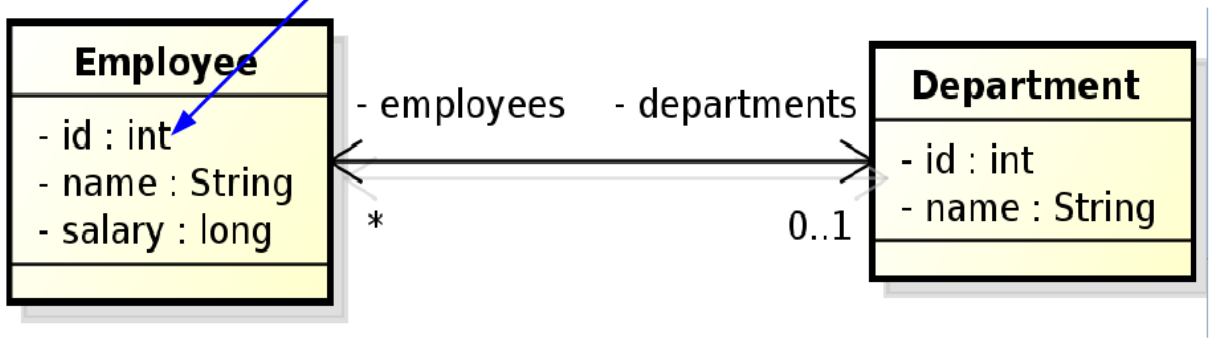


Figure: Keying by entity attribute

Collection Mapping - Maps

```
@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;
    // …
}
```

Figure: Keying by embeddable type

Collection Mapping - Maps

```
@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;
    // …
}
```

Figure: Keying by embeddable type

Collection Mapping - Maps

```
@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")),
}
```

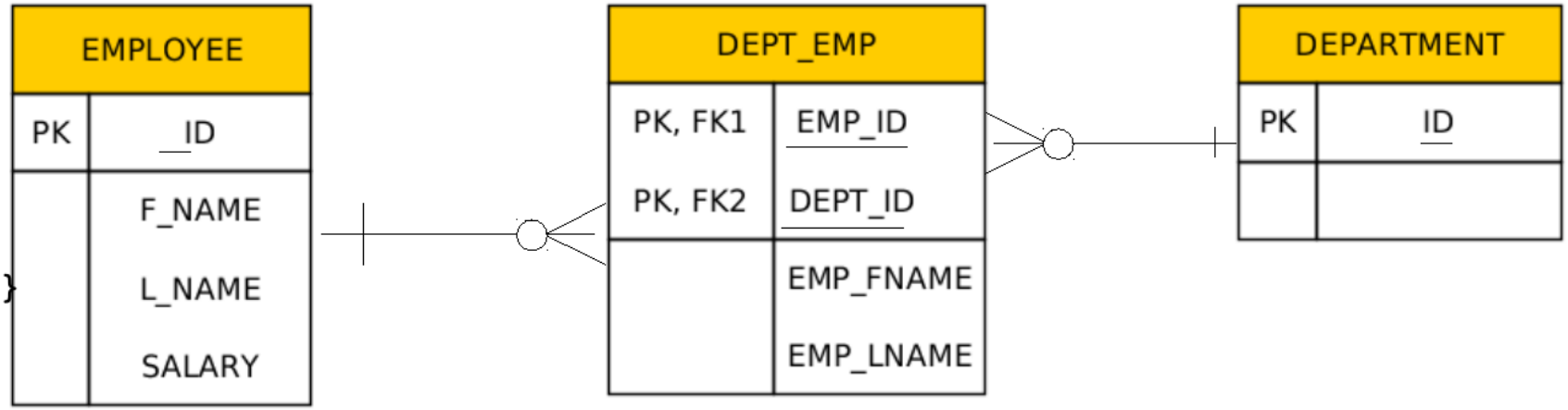


Figure: Keying by embeddable type

Collection Mapping - Maps

```
@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;
    // …
}
```

Figure: Keying by embeddable type

Collection Mapping - Maps

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).

Figure: Keying by embeddable type

Collection Mapping - Maps

```
@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;
    // …
}
```

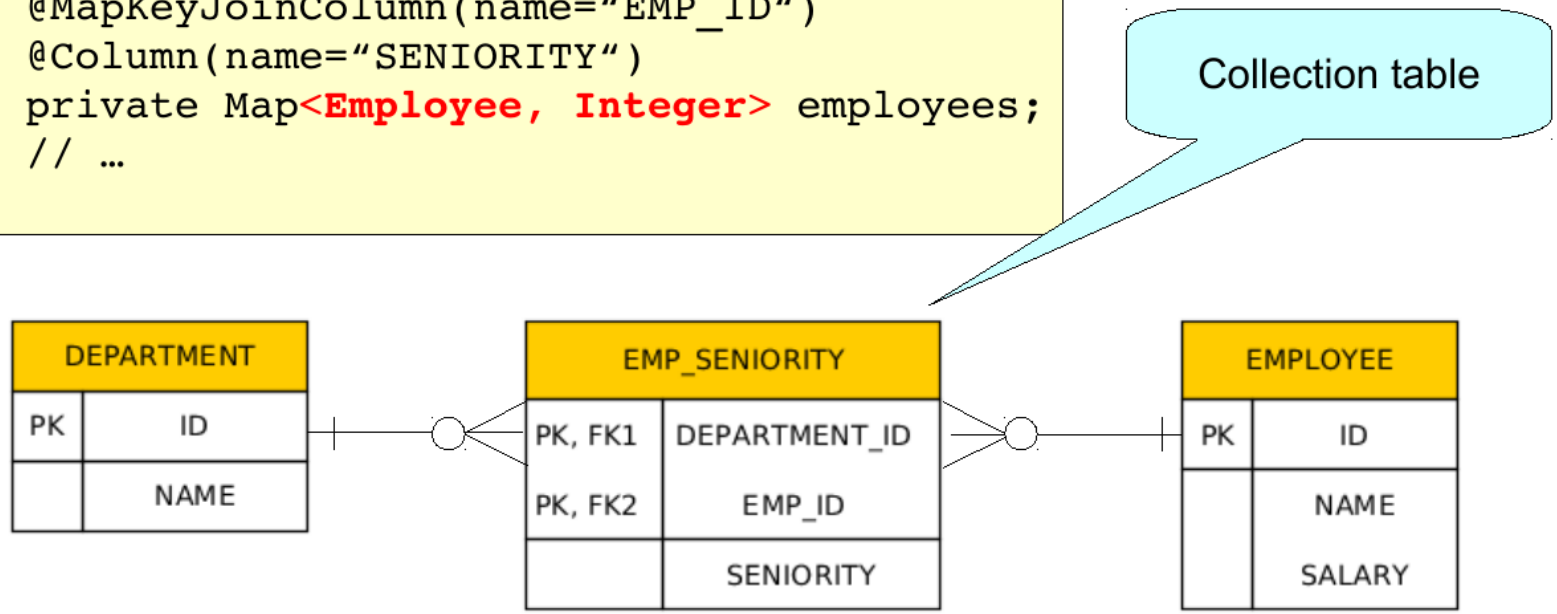


Figure: Keying by entity

Compound and Shared Keys

Compound primary keys

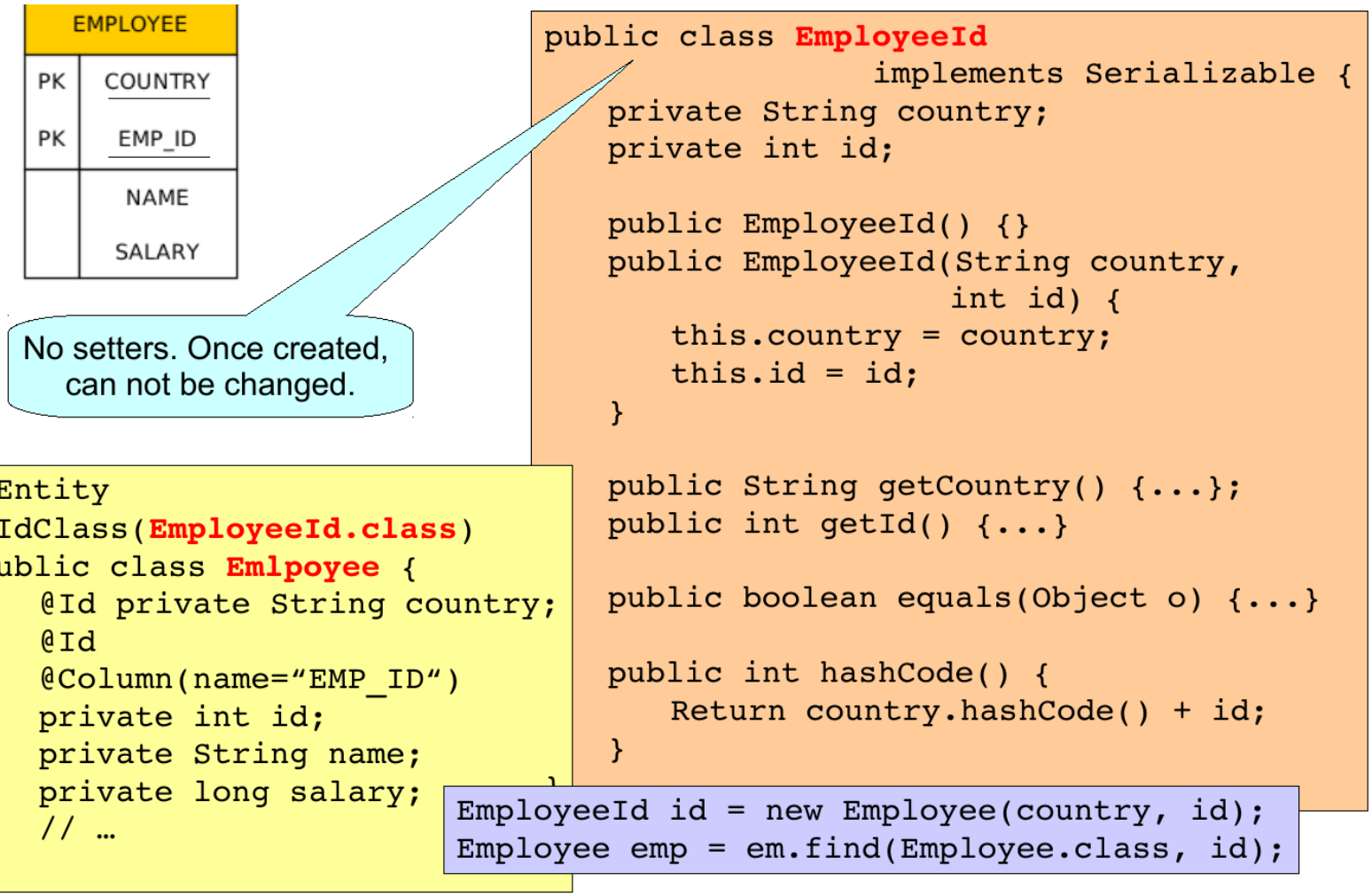


Figure: Id Class

Compound primary keys

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();}
    // …
}
```

Figure: Embedded Id Class

Compound primary keys

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();
```

Figure: Embedded Id Class

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 relationship 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.

Compound Join Columns

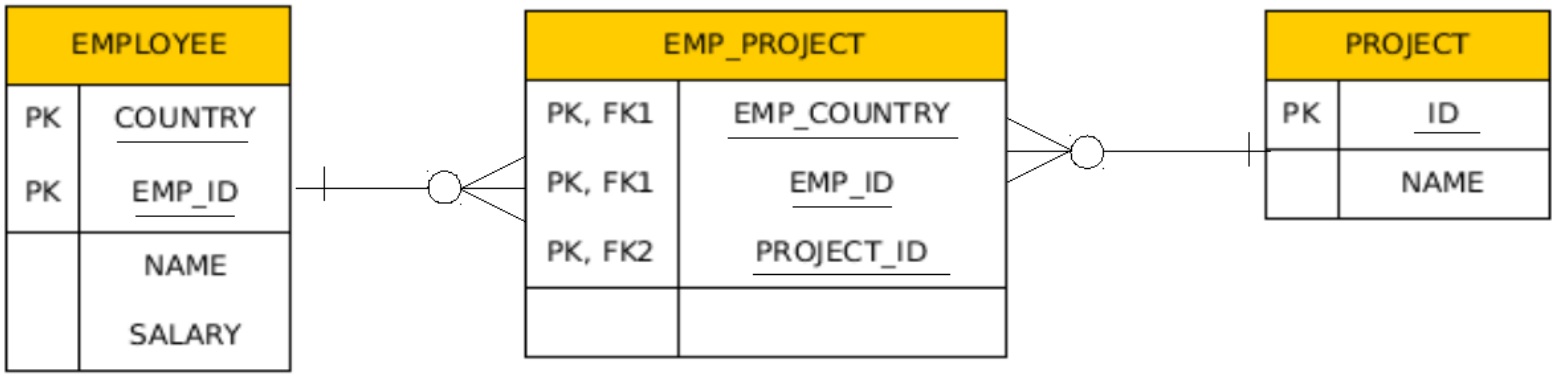
EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY
FK1	MGR_COUNTRY
FK1	MGR_ID

```
@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;
}
```

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;
    // ...
}
```

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")}
    )
```

Criteria API

```
SELECT p FROM Product p WHERE p.name LIKE '%p%'
```

Static Metamodel

```
CriteriaBuilder cb =
  em.getCriteriaBuilder();
CriteriaQuery<Product> cq =
  cb.createQuery(Product.class);
Root<Product> r =
  cq.from(Product.class);
cq.where(
  cb.like(
    r.get(Product_.name)
  ,"%p%")
);
return
  em.createQuery(cq).getResultList();
```

Metamodel

```
Metamodel m = em.getMetamodel();
CriteriaBuilder cb =
  em.getCriteriaBuilder();
CriteriaQuery<Product> cq =
  cb.createQuery(Product.class);
Root<Product> r =
  cq.from(Product.class);
cq.where(
  cb.like(
    r.get(
      m.entity(Product.class)
      .getSingularAttribute("name",
      String.class))
  ,"%p%")
);
return
  em.createQuery(cq).getResultList();
```

Beyond JPA

Graph Databases

- comparing to RDBMS – relation types are **first-class citizens**
- suitable for relaxed data schemas
- suitable for analytics using graph algorithms
- e.g. Neo4j

Spring Data Neo4j

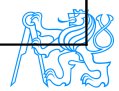
Model:

```
import org.neo4j.ogm.annotation.*;
@NodeEntity
public class Person {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @Relationship(type = "ACTED_IN")
    private List<Movie> movies = new ArrayList<>();
}
```

Repository:

```
@RepositoryRestResource(collectionResourceRel = "movies", path =
  "movies")
public interface MovieRepository extends Neo4jRepository<Movie, Long> {
    @Query("MATCH (m:Movie)<-[r:ACTED_IN]-(a:Person) RETURN m,r,a")
    List<Person> getActors();
}
```



RDF Triple Stores

- comparing to RDBMS – relation types are **first-class citizens**
- suitable for relaxed data schemas
- suitable for Linked Data, semantic web, ontologies
- e.g. RDF4J, Virtuoso, Fuseki, Blazegraph, GraphDB, ...

JOPA is a library for accesing triples using Java objects.

JOPA

Model:

```
import cz.cvut.kbss.jopa.model.annotations.*;
@OWLClass(iri = "http://example.org/ontology/student")
public class Student implements Serializable {

    @Id
    private URI uri;

    @OWLDataProperty(iri = "http://example.org/ontology/email")
    private String email;
}
```

DAO similar to JPA `EntityManager`.

Resources

- JSR 338 Java Persistence 2.1 Final Release
  `http://jcp.org/aboutJava/communityprocess/final/jsr338/index.html`

The End

Thank You