Přístup k databázím, Objektově-relační mapování, JPA 2.0

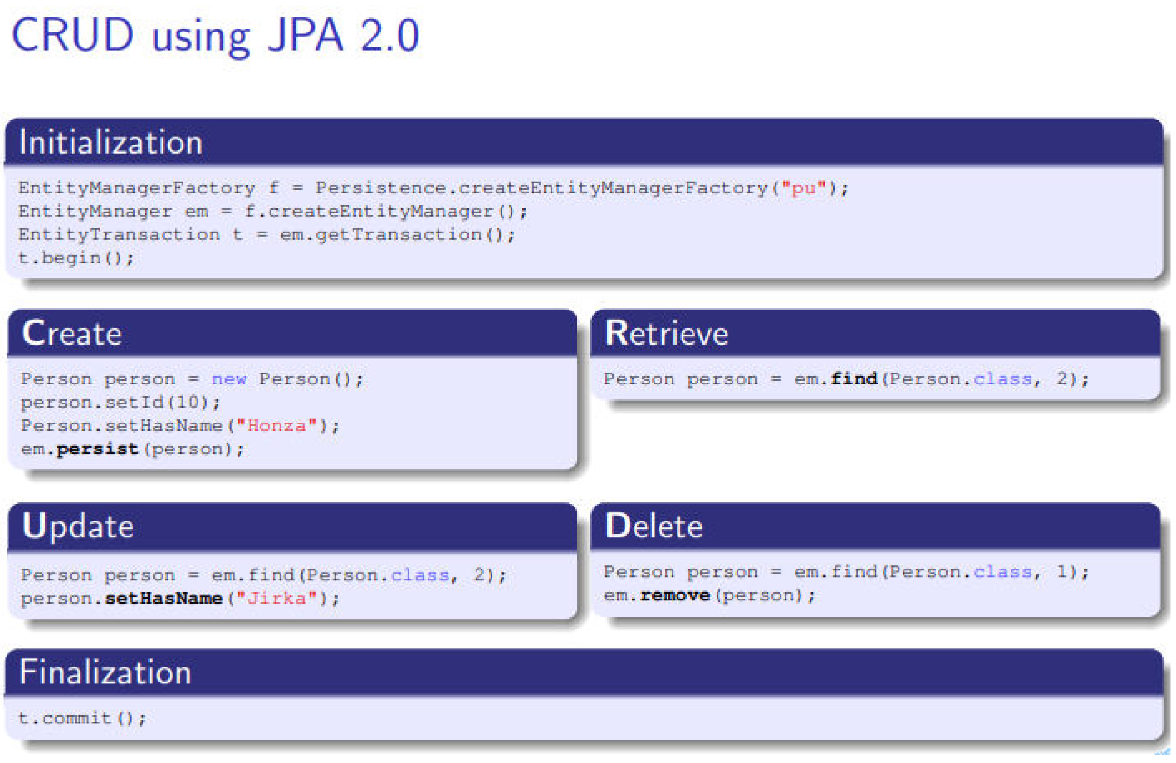
Data persistence

- manipulujeme s daty (reprezentovanými jako stav objektu), které musí být ukládány:
  - persistently – přežít jeden běh aplikace
  - queriably – možnost je načíst / přistupovat k nim
  - scalably – zvládat větší objem dat
  - transactionally – zajistit jejich konzistenci

- Jak jí docílit?
  - serializace - jednoduché, ale ne dotazovatelné ani transakční
  - relační databáze
  - NoSQL databáze
  - RDF Triple Stores

- programátorský přístup k RDBMS:
  - JDBC
    - java standard mající zajistit nezávislost na konkrétním RDBMS
    - prepared statements
  - EJB – nabízí ORM, ale komplikované entity
  - Hibernate – implementace JPA, java framework
  - JPA 2

JPA (Java Persistence API)



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");
```

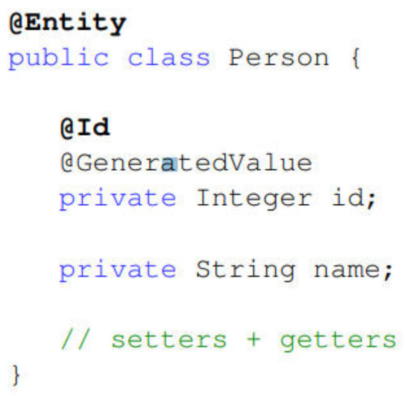
Delete

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

Finalization

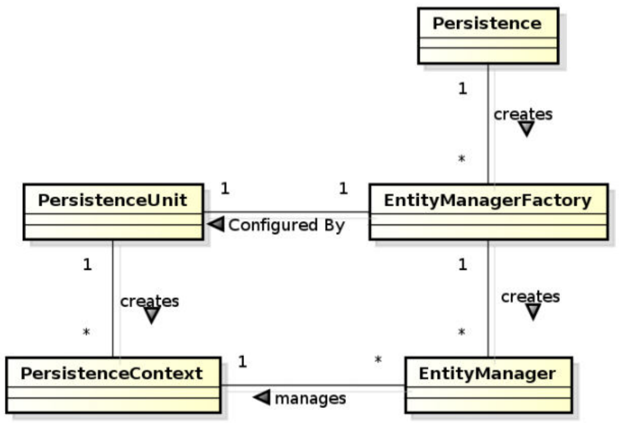
```
t.commit();
```

- Java EE specifikace pro ORM
- součást Java EE 7 specifikací
- vhodně anotované POJOs (entities), popisující doménový model
- set entit je logicky seskupen do persistence unit
- JPA providers
    - generují PU z existující databáze
    - generují DB schéma z existující PU

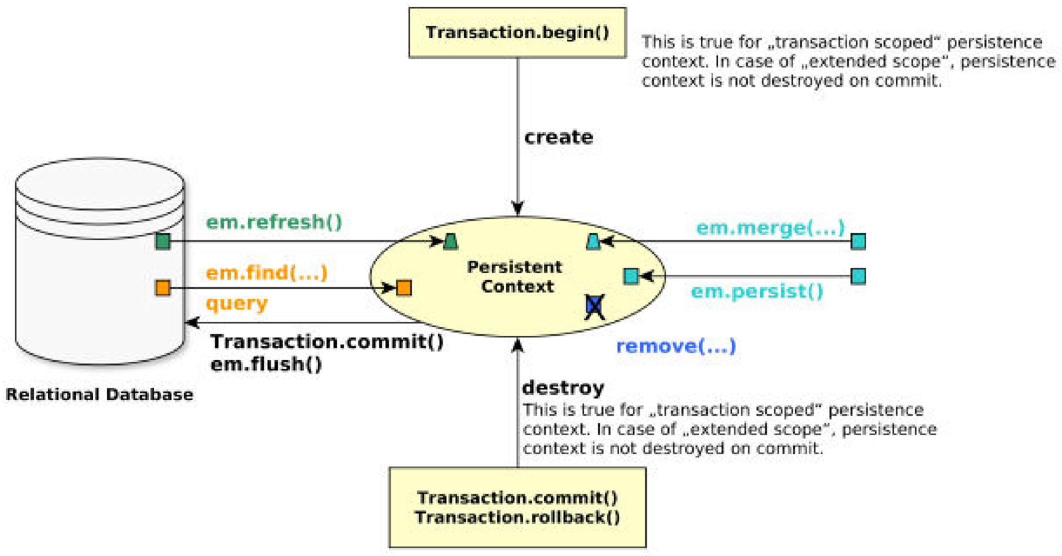


```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    // setters + getters
}
```

1 - minimal example

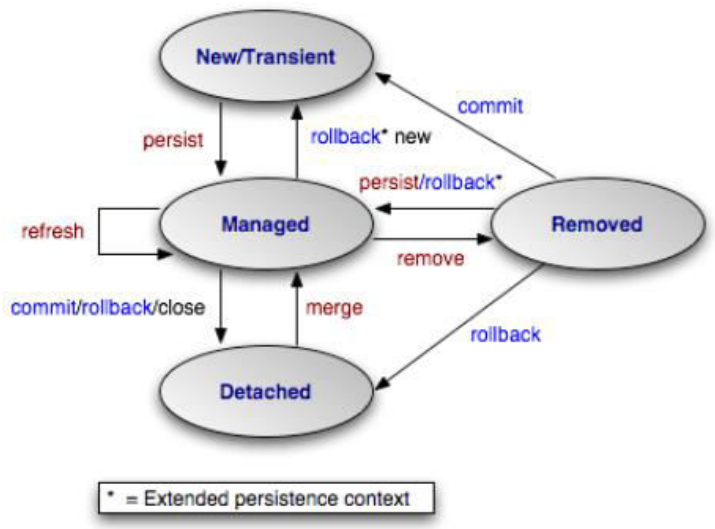


- v runtime aplikace přistupuje k objektu reprezentovanému instancemi entit
- **persistenční kontext** = množina entit, které jsou spravovány jedním správcem entit
    - PC se syncronizuje s DB na vyžádání (refresh, flush) anebo při commitu transakce
    - k PC přistupuje instance EntityManageru, může být sdílen vícerem EMs
    - v jednom PC max. 1 instance entity daného typu se stejným primárním klíčem
    - managed (spravované) = instance entit v PC
    - detached (odpojené) = instance mimo PC



Transaction.begin()
This is true for „transaction scoped" persistence context. In case of „extended scope", persistence context is not destroyed on commit.
create
em.refresh()
em.merge(...)
em.find(...)
query
Persistent Context
em.persist()
Transaction.commit()
em.flush()
remove(...)
Relational Database
destroy
This is true for „transaction scoped" persistence context. In case of „extended scope", persistence context is not destroyed on commit.
Transaction.commit()
Transaction.rollback()

JPA – Entity States



- operace Entity Manageru:
    - **persist** – přidá entitu do kontextu
    - **merge** – mergne detached entitu s manager verzí / upraví entitu v úložišti dle kontextu
    - **find** – načte entitu se zadaným klíčem z úložiště do kontextu
    - **refresh** – obnoví entitu v kontextu dle úložiště
    - **remove** – odebere entitu z úložiště
    - **detach** – odeberu entitu z kontextu

- EntityManager je generické DAO
- CRUD operace na EM:
    - **Create**: em.persist(Object o)
    - **Read**: em.find(Object id), em.refresh(Object o)
    - **Update**: em.merge(Object o)
    - **Delete**: em.remove(Object o)

- nativní dotazy / JPQL: em.createNativeQuery, em.createQuery…
- transakce: em.getTransaction.[begin(),commit(),rollback()]

```
- // vytvořit několik instancí entit
-
- Company company = new Company();
- company.setName("SuperTech a.s.");
-
- Employee employee = new Employee();
- employee.setName("Jan");
- employee.setSurname("Novák");
-
- // vytvořit relaci (nutno svázat obě strany)
-
- company.addEmployee(employee);
-
- // TRANSAKCE
- // =========
```

-
- `EntityManager em = emf.createEntityManager();`
-
- `em.getTransaction().begin();`
-
- `// kontext je prázdný`
-
- `em.persist(company);`
-
- `// kontext: {company}`
-
- `em.persist(employee);`
-
- `// kontext: {company, employee}`
-
- `em.getTransaction().commit();`
-
- `// kontext byl vyprázdněn,`
- `// instance "company" a "employee" jsou odpojené (detached)`

ORM – Objektově-relační mapování

- java třídy = entity = SQL tabulky
- java fields = vlastnosti entity = SQL sloupce
- ORM je realizováno přes Java anotace či XML
- anotace pro fyzické schéma: @Table, @Column, @JoinColumn, @JoinTable,…
- anotace pro logické schéma: @Entity, @OneToMany, @ManyToMany,…
- Each property can be fetched lazily/eagerly.

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

Enums

```
@Enumerated(value=EnumType.String)
private EnumPersonType type;
```

Stored either in a `text` column, or in an `int` column

Temporals

```
@Temporal(TemporalType.Date)
private java.util.Date datum;
```

Identifikátory

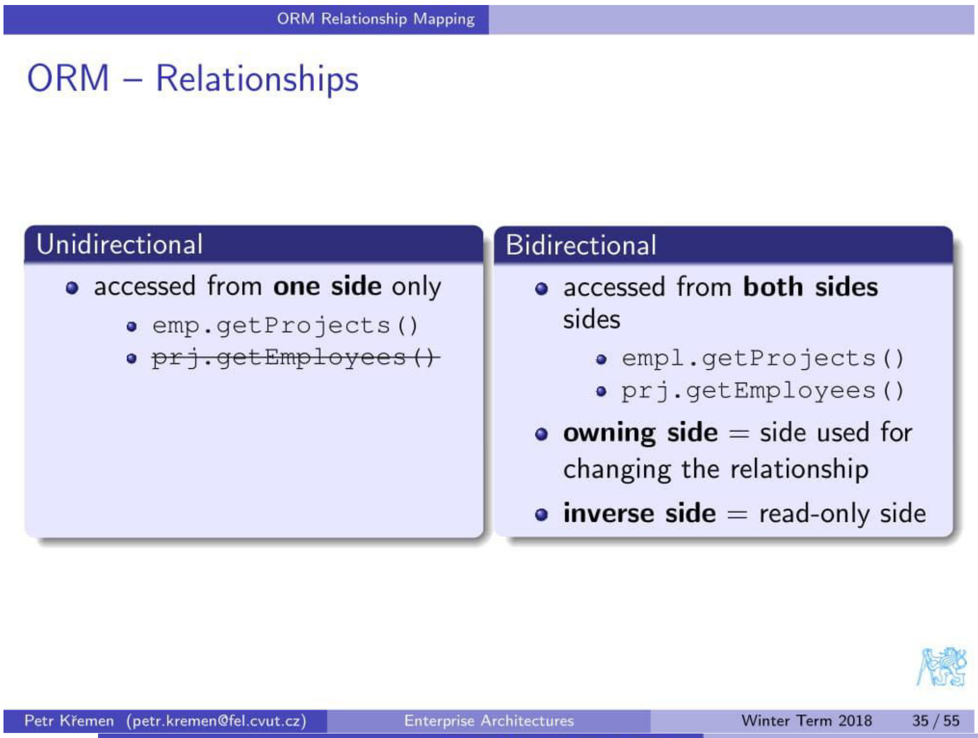
- různé strategie
  - auto
  - table – speciální tabulka pro generované hodnoty
  - sequence – nativní funkce databáze (PostgreSQL) SEQUENCE
  - identity – autonumber sloupce u některých DB
- u DB je hodnota ID nastavena při Transaction.commit(), em.flush() nebo em.refresh()

- Single-attribute: `@Id`
- Multiple-attribute – an identifier class must exist
  - Id. class: `@IdClass`, entity ids: `@Id`
  - Id. class: `@Embeddable`, entity id: `@EmbeddedId`

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;
```

Mapování asociací

- unidirectional – přístup jen z první entity do druhé (ne naopak)
- bidirectional – obě entity o sobě ví
    - owning side = užití pro změnu vztahu
    - inverse side = read-only



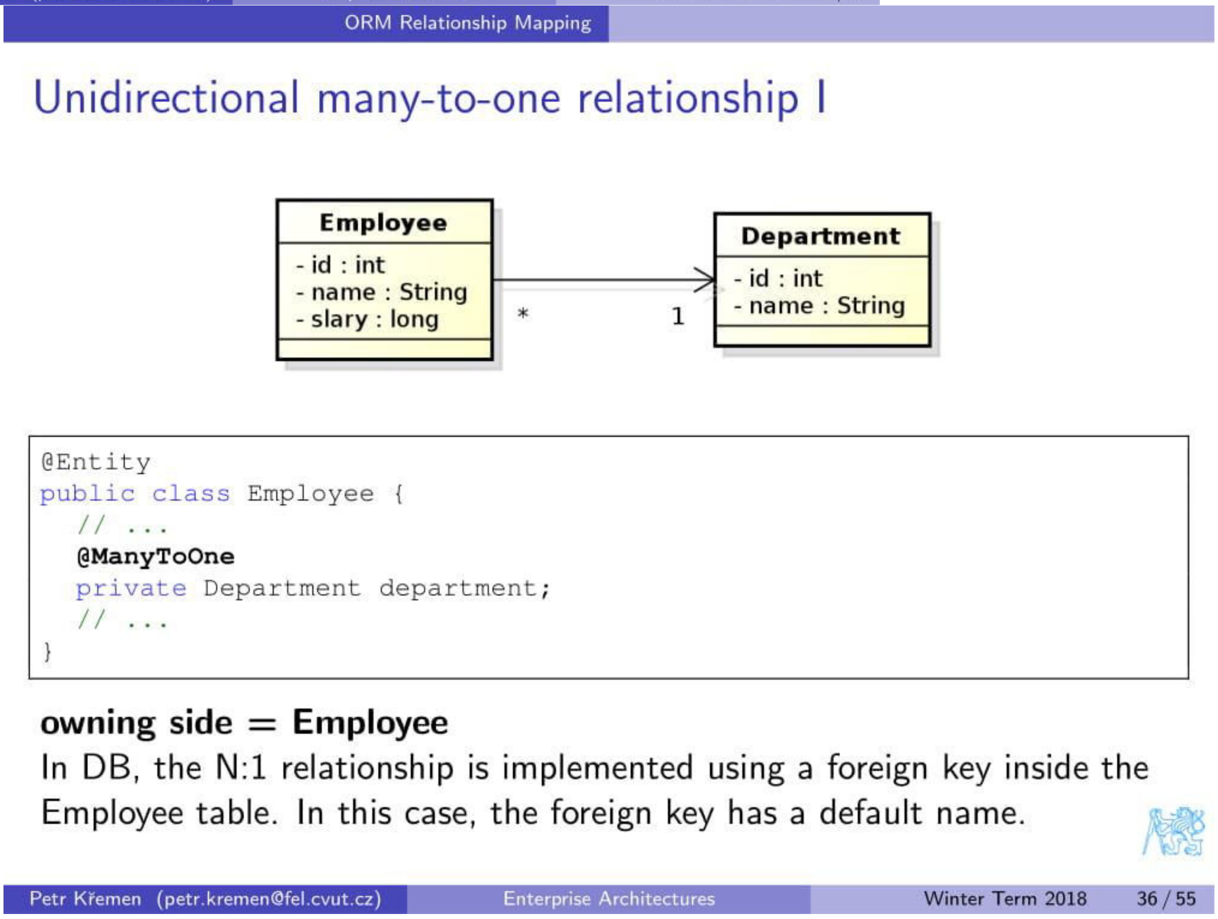
ORM – Relationships

Unidirectional

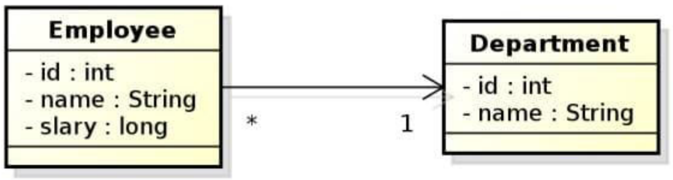
- accessed from **one side** only
    - emp.getProjects()
    - ~~prj.getEmployees()~~

Bidirectional

- accessed from **both sides**
    - empl.getProjects()
    - prj.getEmployees()
- **owning side** = side used for changing the relationship
- **inverse side** = read-only side



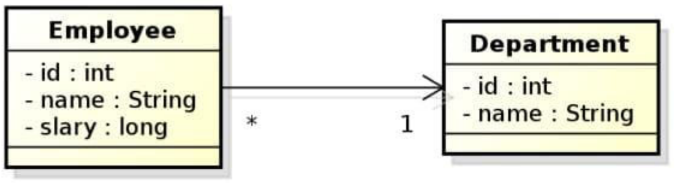
Unidirectional many-to-one relationship I



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

owning side = Employee
In DB, the N:1 relationship is implemented using a foreign key inside the Employee table. In this case, the foreign key has a default name.

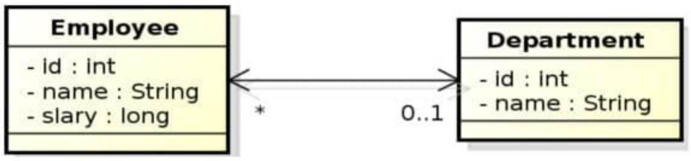
Unidirectional many-to-one relationship II



```
@Entity
public class Employee {
  @Id
  private int id;
  private String name;
  private long salary;
  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
}
```

owning side = Employee.
Here, the foreign key is defined using the `@JoinColumn` annotation.

Bidirectional many-to-one relationship



```
@Entity
public class Employee {
  @Id
  private int id;
  private String name;
  private long salary;
  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
}
```

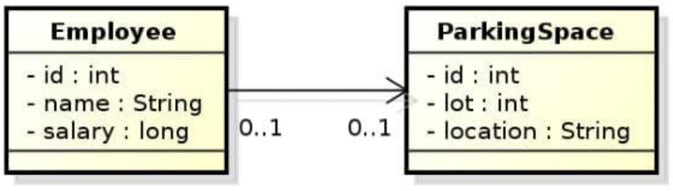
owning side = Employee

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

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

}
```

inverse side = Department

Here, the foreign key is defined using the `@JoinColumn` annotation.

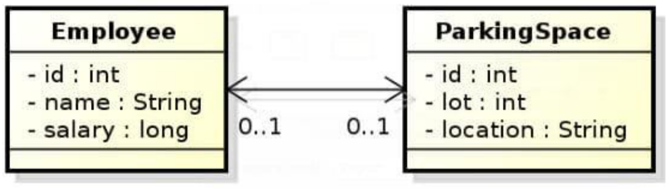
Unidirectional one-to-one relationship



```
@Entity
public class Employee {
  @Id
  private int id;
  private String name;
  private long salary;
  @OneToOne
  @JoinColumn(name="PSPACE_ID")
  private ParkingSpace parkingSpace;
}
```

owning side = Employee.

Bidirectional one-to-one relationship

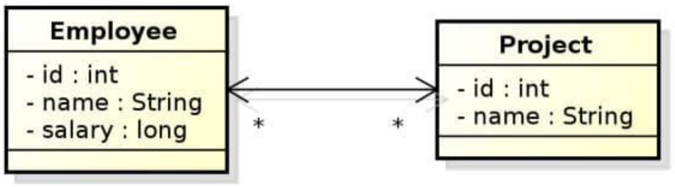


```
@Entity
public class Employee {
  @Id
  private int id;
  private String name;
  private long salary;
  @OneToOne
  @JoinColumn(name="PSPACE_ID")
  private ParkingSpace parkingSpace;
}
```

```
@Entity
public class ParkingSpace {
  @Id
  private int id;
  private int lot;
  private String location;

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

owning side = Employee **inverse side = ParkingSpace**

Bidirectional many-to-many relationship



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

  @ManyToMany
  private Collection<Project>
    project;
}
```

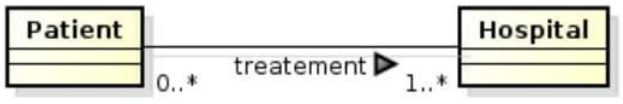
```
@Entity
public class Project {

  @Id private int id;
  private String name;

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

inverse side = ParkingSpace

owning side = Employee

Conceptual Modeling Intermezzo

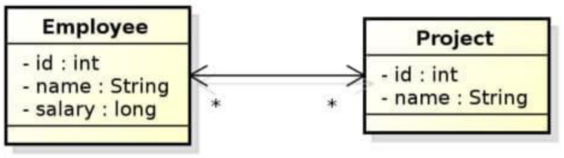
- M:N relationship is a **conceptual modeling** primitive



- Does it mean that
  - A patient has **one** treatment that is handled in **more** hospitals ?
  - A patient has **more** treatments, each handled in a **single** hospital ?
  - A patient has **more** treatments, each handled in **more** hospitals ?
- partialities and cardinalities are too weak in this case.

Careful modeling often leads to decomposing M:N relationships on the **conceptual level** (not on the logical level, like JPA).

Bidirectional many-to-many relationship

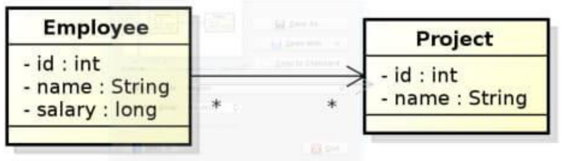


```
@Entity
public class Employee {
  @Id private int id;
  private String Name;
  private long salary;
  @ManyToMany
  @JoinTable(name="EMP_PROJ",
    joinColumns=
      @JoinColumn(name="EMP_ID"),
    inverseJoinColumns=
      @JoinColumn(name="PROJ_ID"))
  private Collection<Project>
    projects;
}
```

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

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

inverse side = ParkingSpace

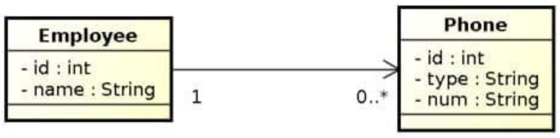
Unidirectional many-to-many relationship



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

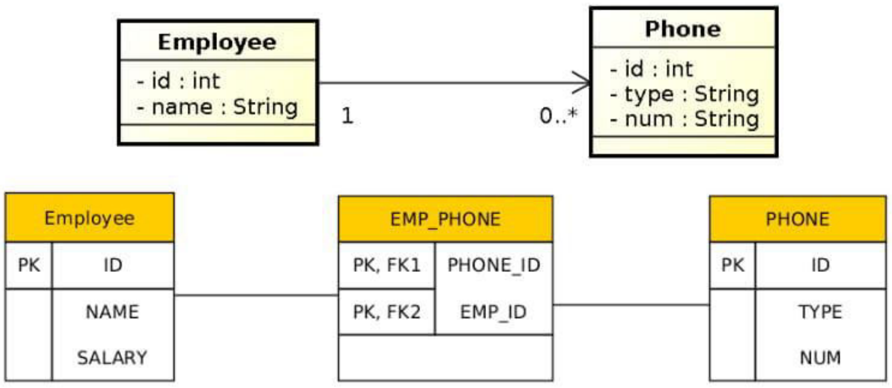


```
@Entity
public class Employee {
  @Id private int id;
  private String name;
  @OneToMany
  @JoinTable(name="EMP_PHONE",
    joinColumns=
      @JoinColumn(name="EMP_ID"),
    inverseJoinColumns=
      @JoinColumn(name="PHONE_ID"))
  private Collection<Phone> phones;
}
```

```
@Entity
public class Phone {
  @Id private int id;
  private String type;
  private String num;
}
```

owning side = Employee

Unidirectional one-to-many relationship



```
@Entity public class Employee {
  @Id private int id;
  private String name;
  @OneToMany @JoinTable(name="EMP_PHONE",
    joinColumns=@JoinColumn(name="EMP_ID"),
    inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
  private Collection<Phone> phones;
}
```

```
@Entity
public class Phone {
  @Id private int id;
  private String type;
  private String num;
}
```

Lazy Loading

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


 private ParkingSpace
   parkingSpace;
}
```

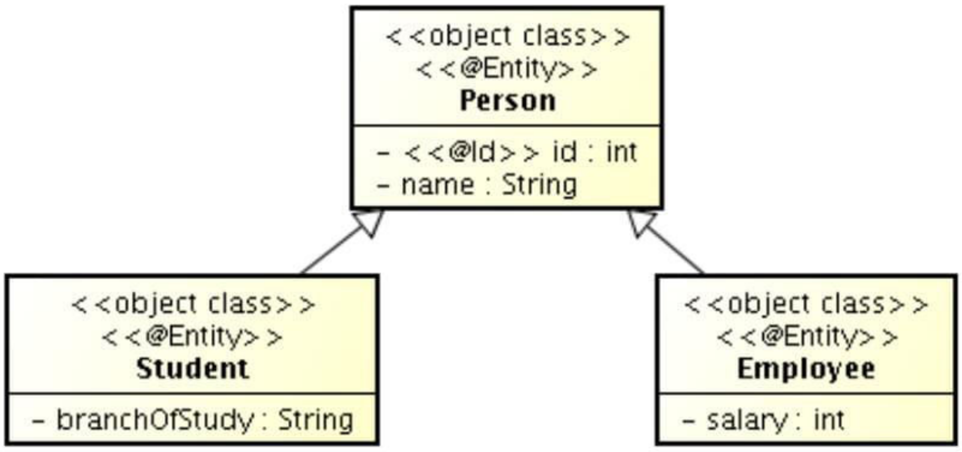
```
@Entity
public class Employee {
 @Id private int id;
 private String name;

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

`parkingSpace` instance fetched from the DB at the time of reading the `parkingSpace` field.

ORM Inheritance Mapping

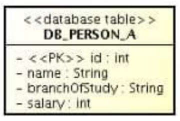
Inheritance

How to map inheritance into DB ?



Strategies for inheritance mapping

single table



joined

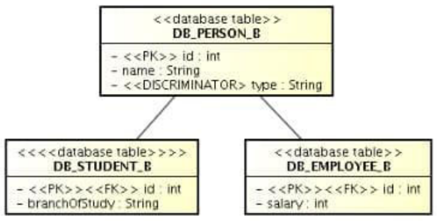
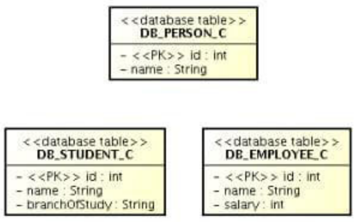


table per class



Inheritance mapping (single-table)

```
@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)

```
@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-class)

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Person { ... }

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

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