Accessing Database System

Martin Řimnáč

Accessing Database System

- client - server architecture
- heterogeneous data types and data structure
- applications can use many data sources
- need to standardize
  - ODBC (Open DataBase Connectivity)
    - an interface for managing connection, authorization, query and result delivery
    - in java: JDBC
  - ORM (Object Relational Mapping)
    - direct usage relational data in object oriented programming
    - In java: JPA (Java Persistence API)

ODBC (Open Database Connection)

- makes an application independent on
  - Database Management System and its version
  - operating system (enable ports to various plarforms)
- introduced in MS Windows in early 1990s
- steps:
  - create connection
  - create (prepared) statement
  - execute statement
  - result browsing
  - closing connection

ODBC – create a connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SlonDataTutorial {
    protected Connection connection;
    protected function createConnection (connectionString, userName, userPassword)
    {
        Class.forName("org.postgresql.Driver");
        this.connection = DriverManager.getConnection(connectionString,userName,userPassword);
    }
    public function execute ()
    {
        try {
                this.createConnection ("jdbc:postgresql://slon.felk.cvut.cz:5432/tutorialexample",
                                                        "tutoruser", "tutorpass");
        } catch (ClassNotFoundException e) {
            System.out.println("PostgreSQL JDBC driver not found."); e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Connection failure."); e.printStackTrace();
        }
    }
}
```

ODBC – create a connection

- The ODBC driver is dynamically loaded
  - Class.forName("org.postgresql.Driver");
  - provides DriverManager
    - creates Connection
- The connection parameters are concerned into the connection string
  - "jdbc:postgresql://slon.felk.cvut.cz:5432/tutorialexample",
    - Means: *Postgresql* DBMS, server *slon.felk.cvut.cz*, port *5432*, database *tutorialexample*
- Authorization
  - Database user *tutoruser* authorized by password *tutorpass*

ODBC – execute a query

- The ODBC connection creates a Statement
    - the statement is executed by the query
    - returns ResultSet
- The ResultSet is navigated by next() method
    - Fields are accessed by the field names or position

```
protected function getAllItems ()
{
    Statement statement = this.connection.createStatement();
    ResultSet resultSet = statement.executeQuery("SELECT model, price FROM item");
    while (resultSet.next()) {
        this.printItem (resultSet);
    }
}
protected function printItem (ResultSet resultSet)
{
   System.out.printf("%-30.30s  %-30.30s%n", resultSet.getString("model"), resultSet.getFloat("price"));
}
```

ODBC – execute query

- The queries can be parametrized
    - SQL Query is constructed as the string concatenation

```
protected function getItemById (int id)
{
  Statement st = this.connection.createStatement();
  ResultSet rs = st.executeQuery("SELECT model, price FROM item WHERE id_item = " + id);
  while (rs.next()) {
        this.printItem(rs);
  }
  rs.close();
  st.close();
}
```

ODBC – execute query

- The queries can be parametrized
  - SQL Query is contructed as the string concatenation

```
protected function getItemById (int id)
{
  Statement st = this.connection.createStatement();
  ResultSet rs = st.executeQuery("SELECT model, price FROM item WHERE id_item = " + id);
  while (rs.next()) {
        this.printItem(rs);
  }
  rs.close();
  st.close();
}
```

- Never use this query construction – SQL Injection

ODBC – SQL injection

```
protected function authorizeEndUser (String userName, String userPassword)
{
  Statement st = this.connection.createStatement();
  ResultSet rs = st.executeQuery("SELECT * FROM authorizeduser
      WHERE username=‘“ + userName + „‘ AND password = ‘“ + userPassword +“‘“
  if (rs.next()) {
       this.setAuthorizedUser(rs);
  }
  rs.close();
  st.close();
}
```

- Authorization example
  - Works perfect for userName=“user“ and userPassword=“heslo“

ODBC – SQL injection

```
protected function authorizeEndUser (String userName, String userPassword)
{
   Statement st = this.connection.createStatement();
   ResultSet rs = st.executeQuery("SELECT * FROM authorizeduser
        WHERE username=‘“ + userName + „‘ AND password = ‘“ + userPassword +“‘“
   if (rs.next()) {
         this.setAuthorizedUser(rs);
   }
   rs.close();
   st.close();
}
```

- Authorization example
  - Works perfect for userName=“user“ and userPassword=“heslo“
  - Fails for userPassword=“‘ OR ‘“=‘“ because the value changes the statement
    - Username = ‘user‘ AND password=‘‘ OR ‘‘=‘‘
- Required to use the prepared statement

ODBC – prepared statement

- The query is initialised as a pattern
  - Query parametrization is provided by ?
  - setInt, setString, … method for value substitution
  - Functionality: the parameter does not change the query

```
protected function getItemByIdSave (int idItem)
{
   PreparedStatement st = this.connection.prepareStatement
                                     ("SELECT model, price FROM item WHERE id_item = ?");
   st.setInt(1, idItem);
   ResultSet rs = st.executeQuery();
   while (rs.next()) {
            this.printItem (rs)
    }
    rs.close();
    st.close();
}
```

Best Practice

- Do not use * in SELECTs, if not necesary
  - Eliminates transfer of unused attributes
- Use data paging (LIMIT/OFSET)
  - Nobody wants to see all the items
  - Required also the result to be sorted
- Do not export artificial keys (id_…)
  - Use the key value corresponding to reality in all external APIs (interoperability)
- Do not use INSERTs without attributes
  - Application is not resistent to data schema changes

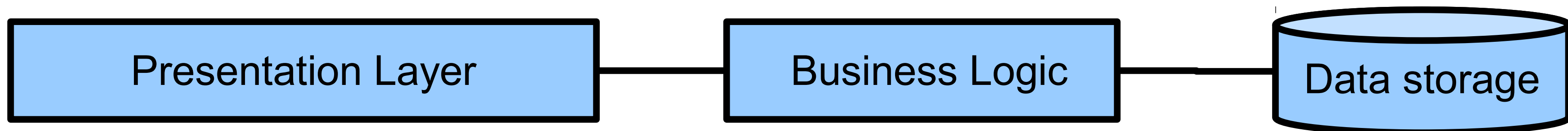
laboratory Gerstner

ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

What is Object-relational mapping ?

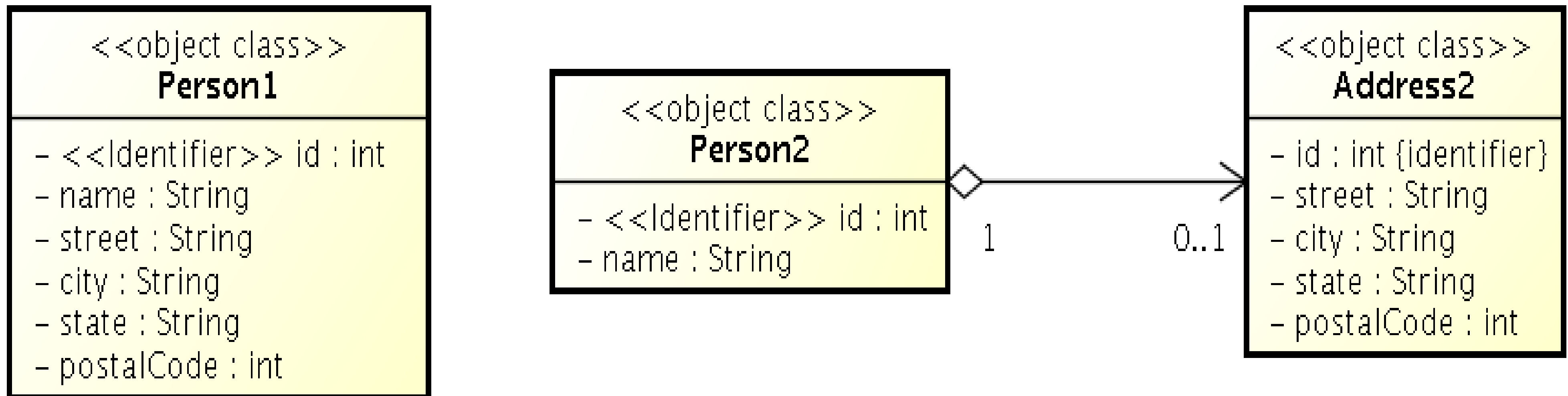
- a typical information system architecture:



- How to avoid data format transformations when interchanging data from the (OO-based) presentation layer to the data storage (RDBMS) and back ?
- How to ensure persistence in the (OO-based) business logic ?

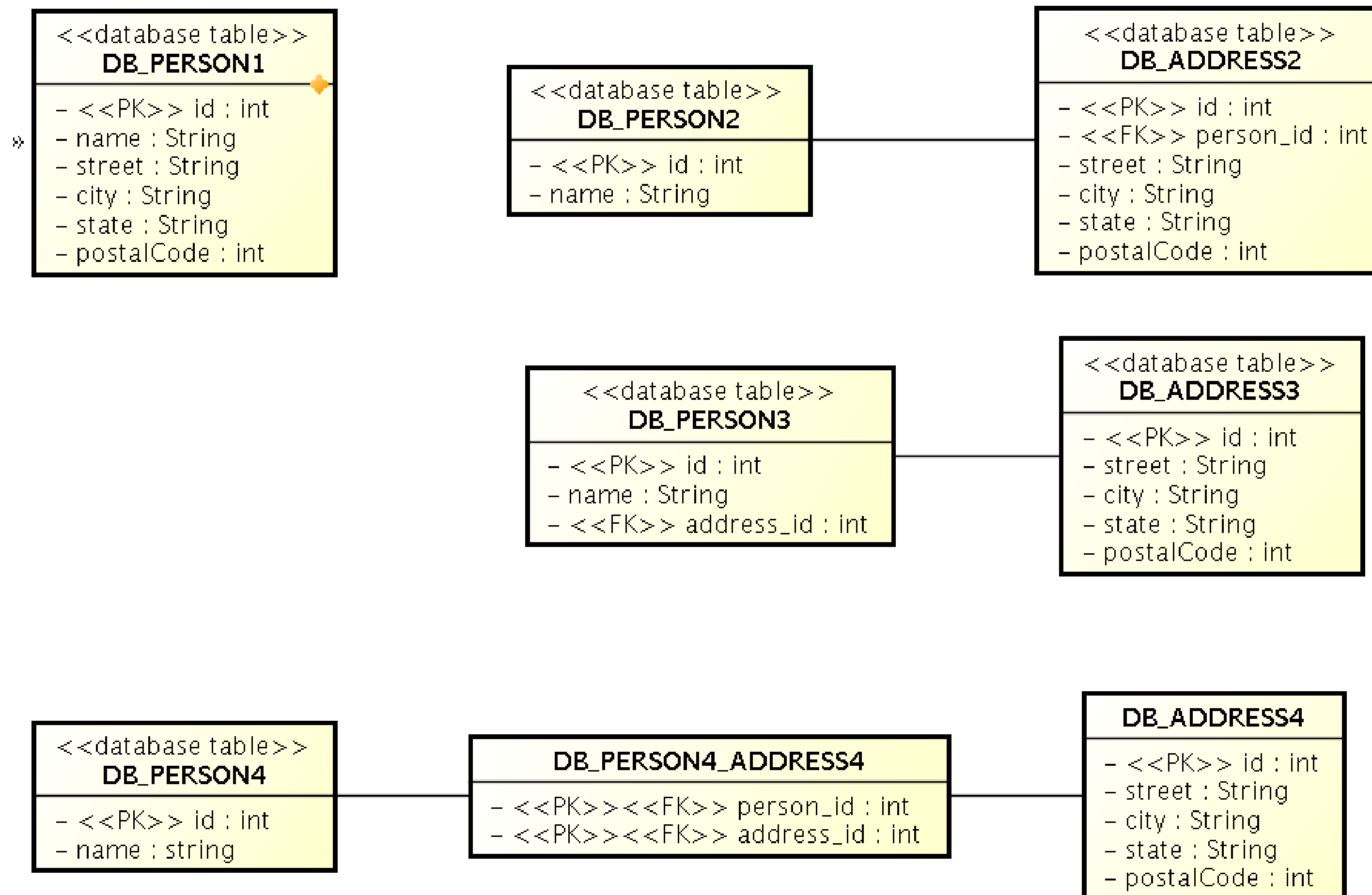
Example – object model

- When would You stick to one of these options ?



Example – database

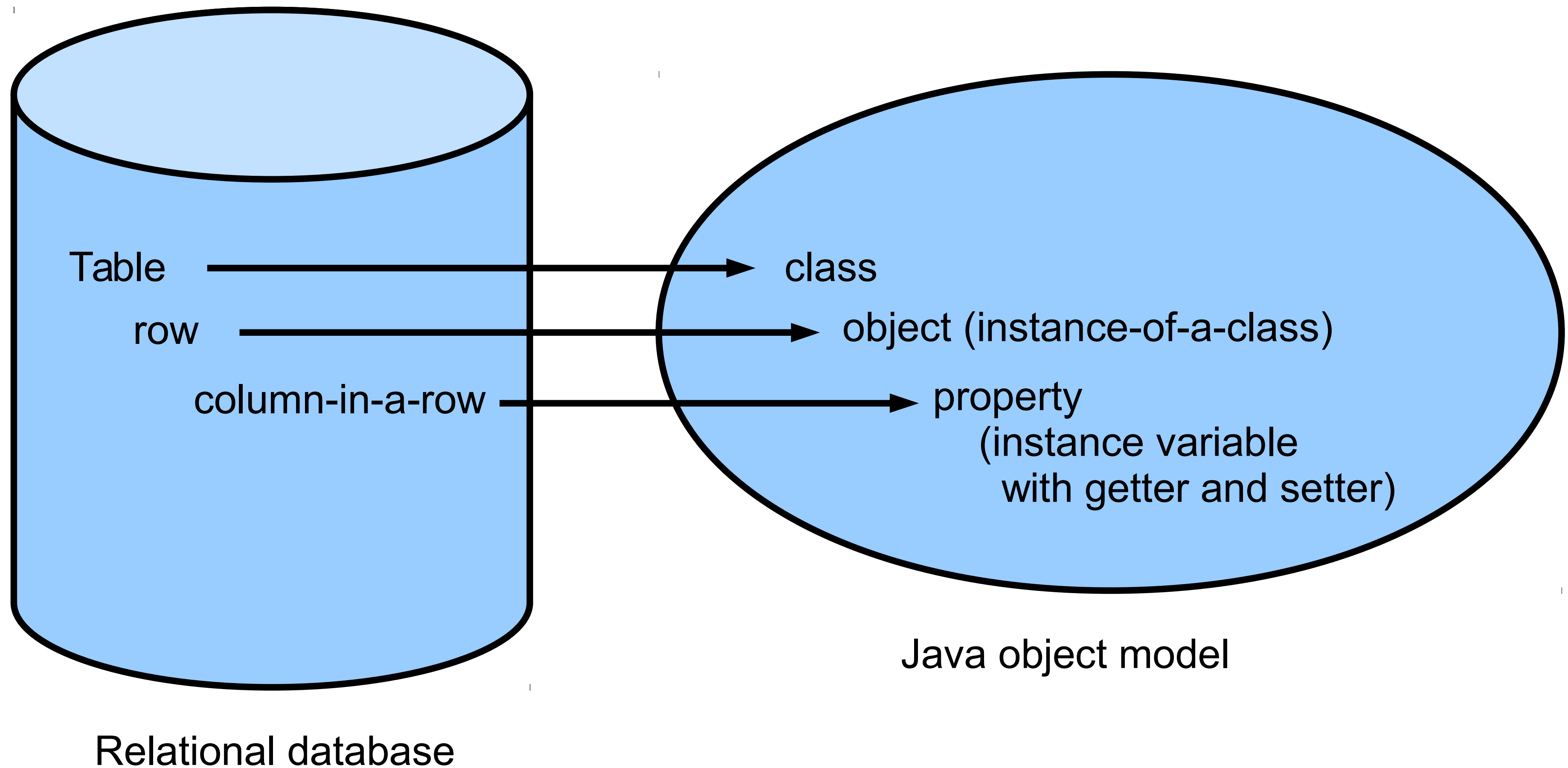
- … and how to model it in SQL ?



Object-relational mapping

- Mapping between the database (declarative) schema and the data structures in the object-oriented language.
- Let's take a look at JPA 2.0

Object-relational mapping



JPA 2.0

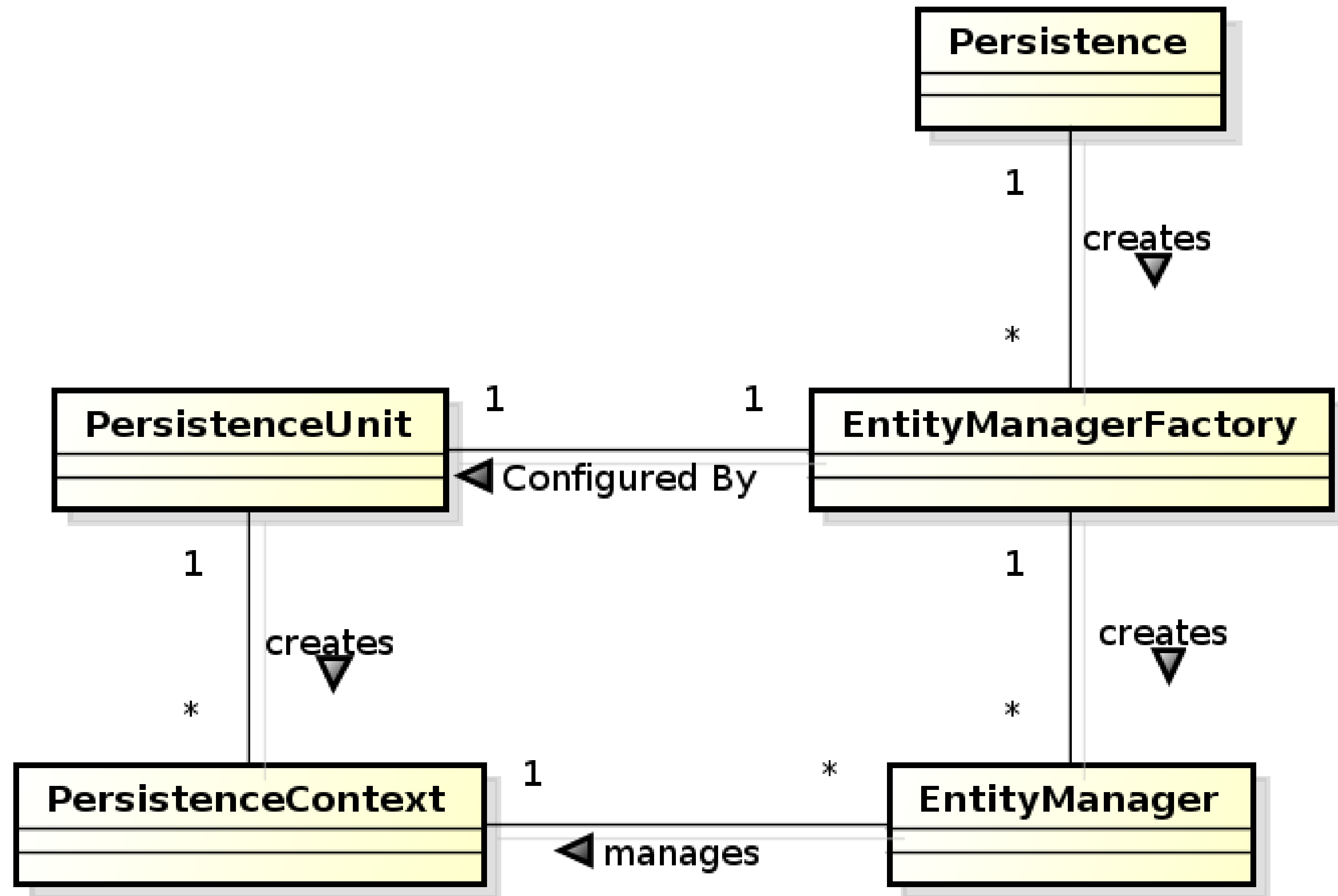
- Java Persistence API 2.0 (JSR-317)
- Although part of Java EE 6 specifications, JPA 2.0 can be used both in EE and SE applications.
- Main topics covered:
  - Basic scenarios
  - Controller logic – `EntityManager` interface
  - ORM strategies
  - JPQL + Criteria API

JPA 2.0 – Entity Example

- Minimal example (configuration by exception):

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

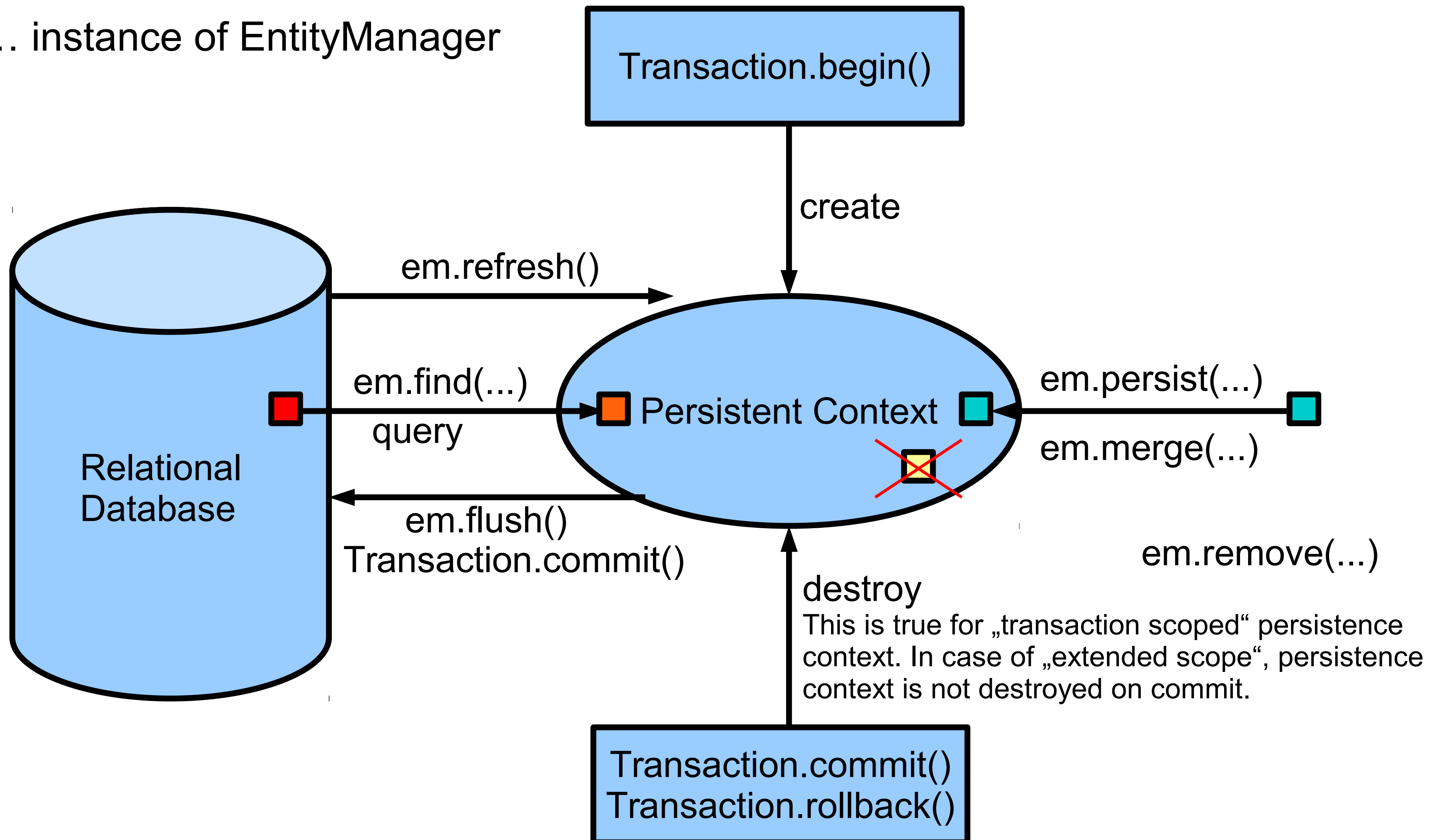
JPA2.0 – Basic concepts



JPA 2.0 - Basics

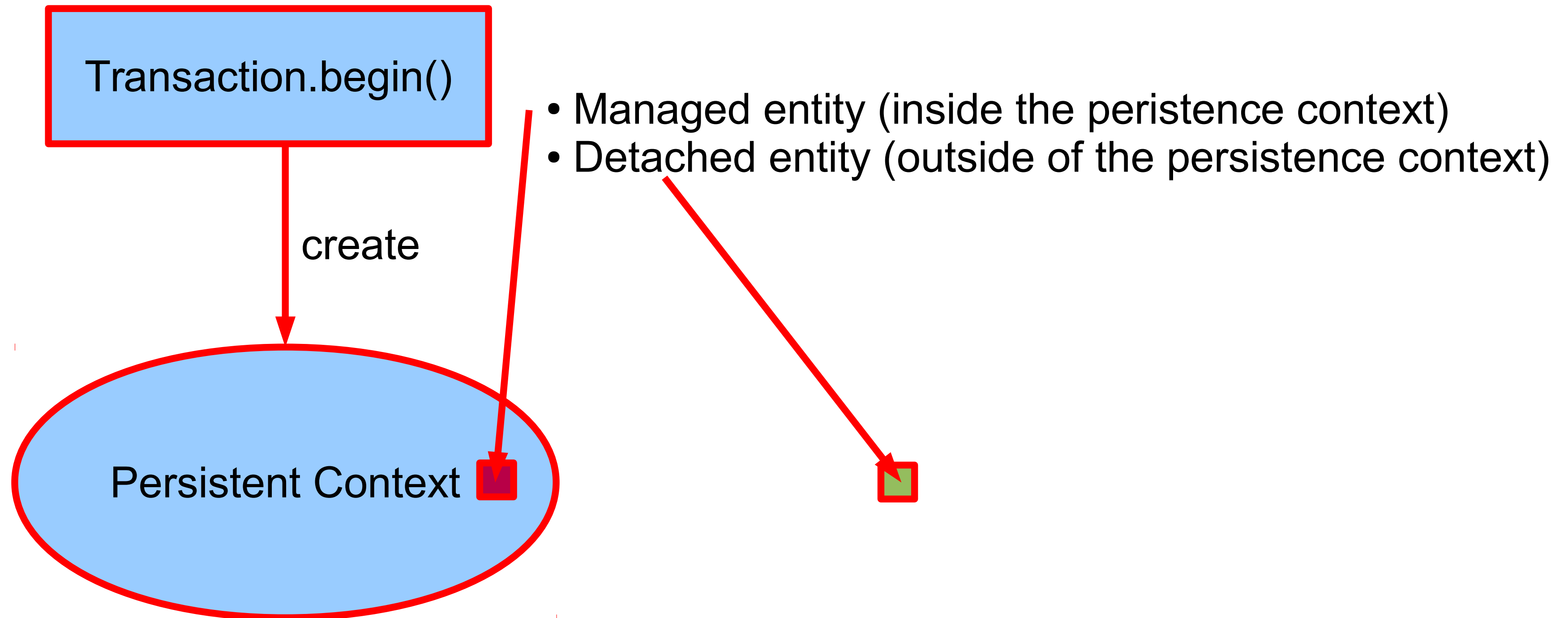
- Let's have a set of „suitably annotated“ POJOs, called ***entities***, describing your domain model.
- A set of entities is logically grouped into a ***persistence unit***.
- JPA 2.0 providers :
  - generate persistence unit from existing database,
  - generate database schema from existing persistence unit.
  - TopLink (Oracle) … JPA
  - EclipseLink (Eclipse) … JPA 2.0
- 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 – 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

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: createNativeQuery, createQuery, etc.
  - Resource-local transactions: getTransaction().[begin(),commit(),rollback()]

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.

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 setId(int id) {this.id=id;}
   …
}
```

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

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 within the 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;
}
```

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 (nullable, insertable, updatable, precise data type, defaults, etc.)
- @Lob – large objects
- Default EAGER fetching (except Lobs)

ORM – Enums, dates

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

  `private EnumPersonType type;`

  - Stored either in text column, or in 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

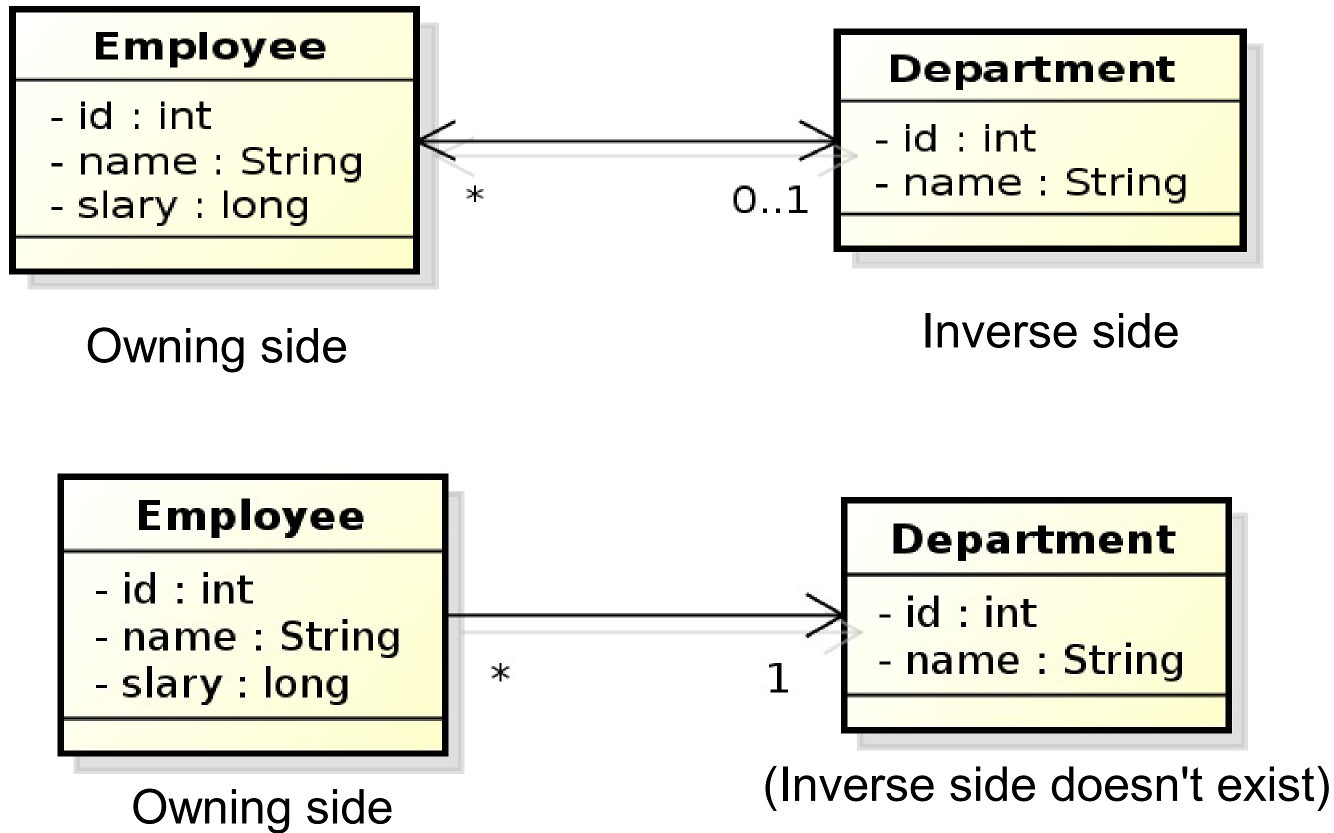
Strategies

- 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

Generated Identifiers TABLE strategy

```
@TableGenerator(
    name="AddressGen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    pkColumnValue="ADDR_ID",
    initialValue=10000,
    allocationSize=100)

@Id @GeneratedValue(generator="AddressGen")

private int id;
```

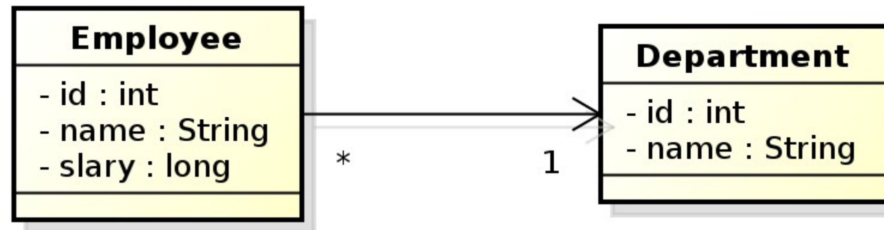
ORM – Relationships



ORM – Relationships

		unidirectional	bidirectional
many-to-one	owning	@ManyToOne [@JoinColumn]	@ManyToOne [@JoinColumn]
	inverse	X	@OneToMany(mappedBy)
one-to-many	owning	@OneToMany [@JoinTable]	@OneToMany [@JoinColumn]
	inverse	X	@ManyToOne(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

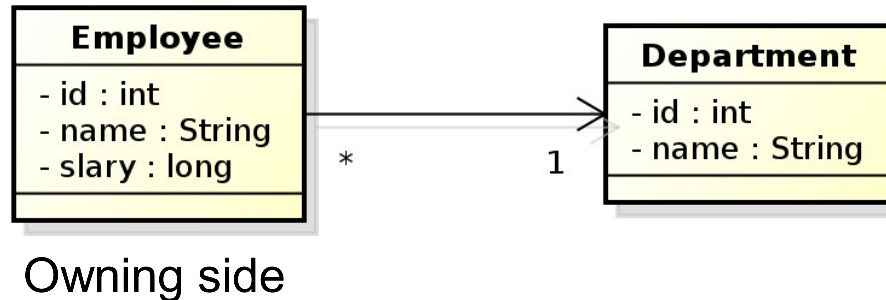


Owning side

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

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

Unidirectional many-to-one relationship

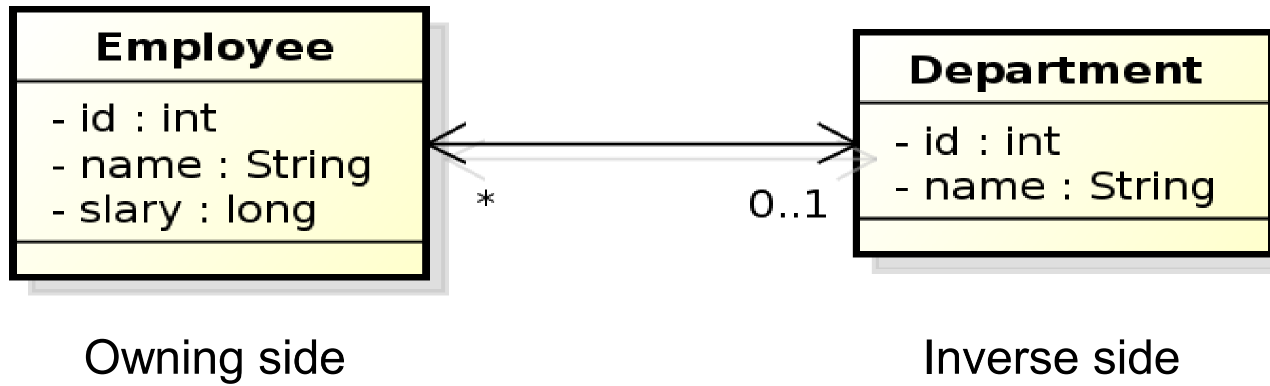


Owning side

```
@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 as the @JoinColumn annotation.

Bidirectional many-to-one relationship



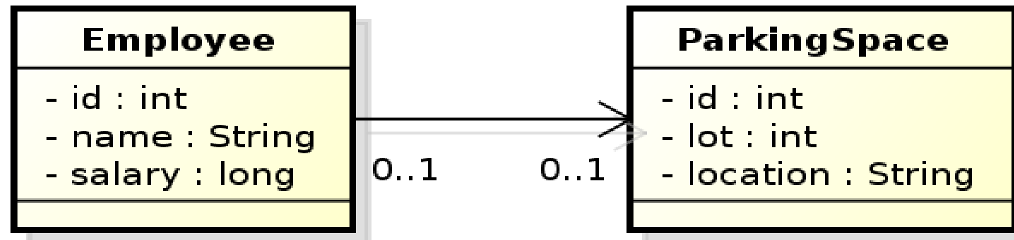
Owning side

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;
}
```

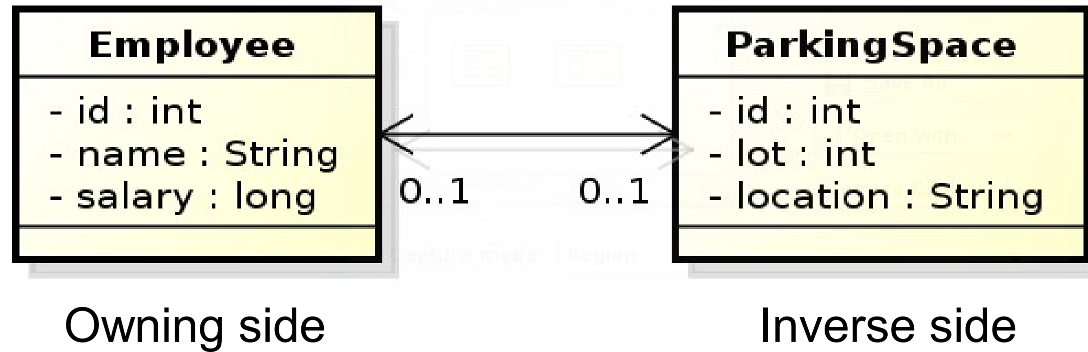
Unidirectional one-to-one relationship



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



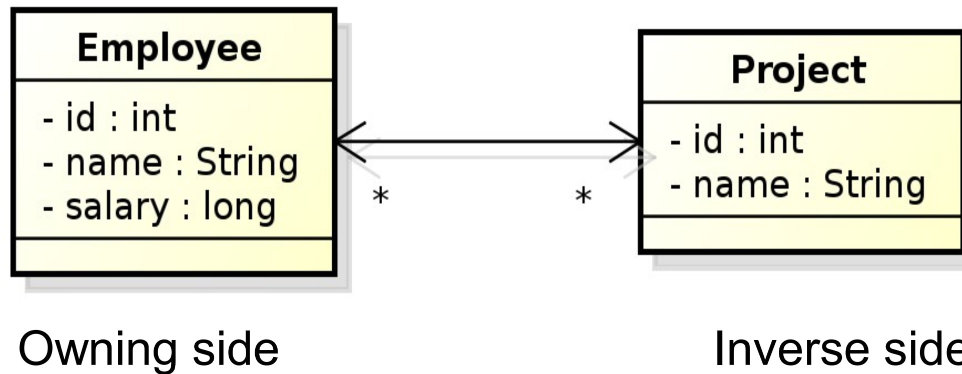
Owning side

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

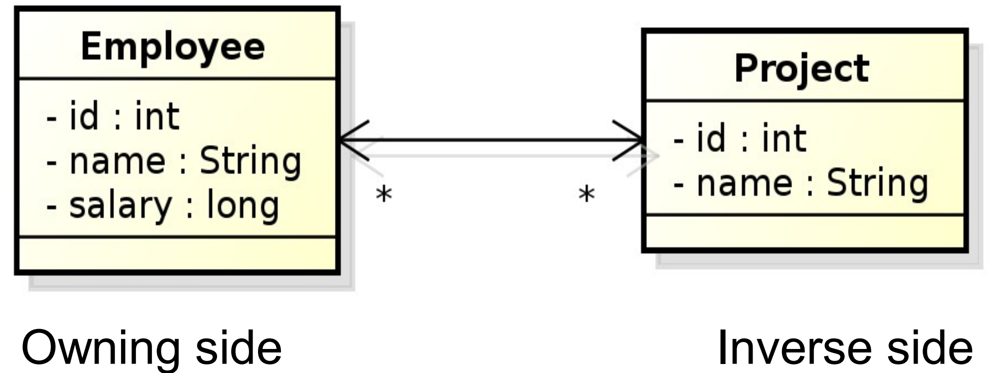
Inverse side

```
@Entity
public class Employee {

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

}
```

```
@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.

Bidirectional many-to-many relationship



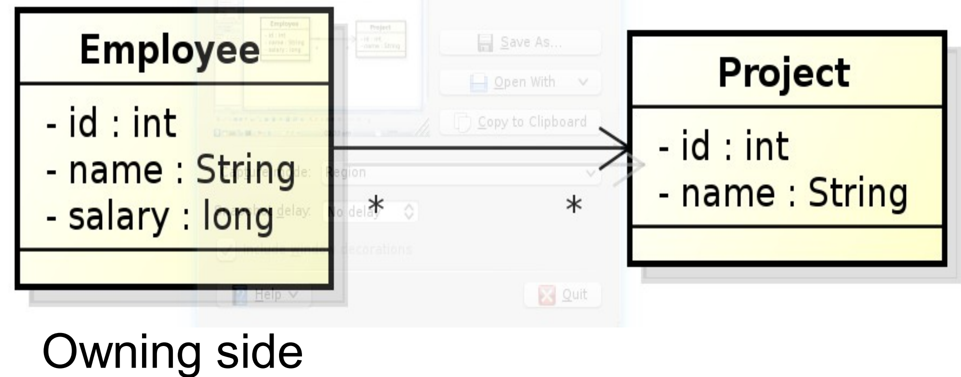
Owning side

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> project;

}
```

```
@Entity
public class Project {

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

}
```

Unidirectional many-to-many relationship



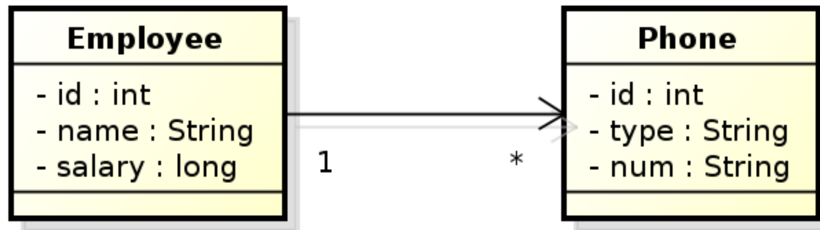
Owning side

```
@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> project;

}
```

```
@Entity
public class Project {

    @Id private int id;
    private String name;

}
```

Unidirectional one-to-many relationship

JPA 2.0 spec: *The **default** mapping for unidirectional one-to-many relationships uses a **join table**. Unidirectional one-to-many relationship **may be** implemented using **one-to many foreign key mappings**, using the JoinColumn and JoinColumns annotations.*



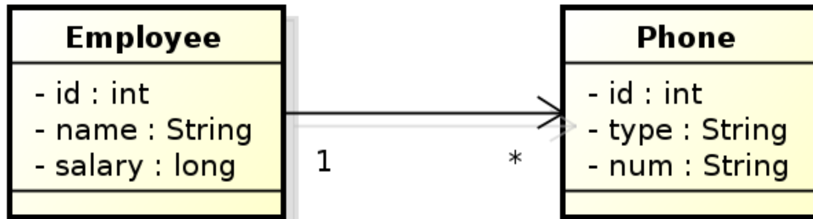
Owning side

```
@Entity
public class Employee {

    @Id private int id;
    private String name;
    private float salary;
    @OneToMany
    @JoinColumn(name="EMP_ID")
    private Collection<Phone> phones;

}
```

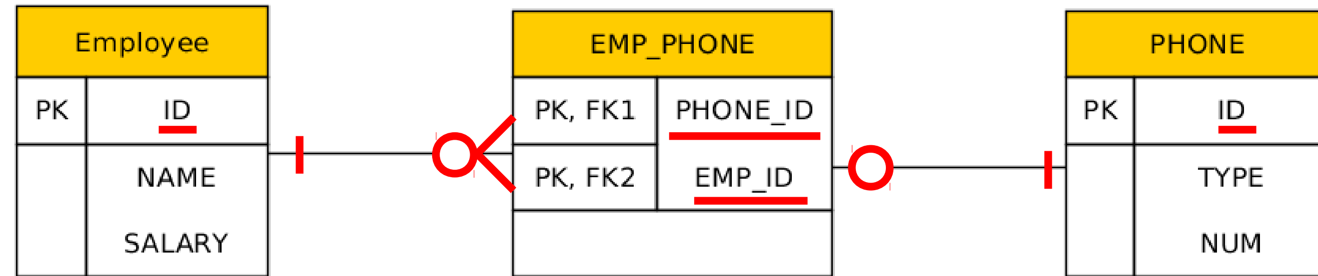
Not available prior to JPA 2.0

// join column is in the table for Phone

Unidirectional one-to-many relationship



Owning side



Logical database schema

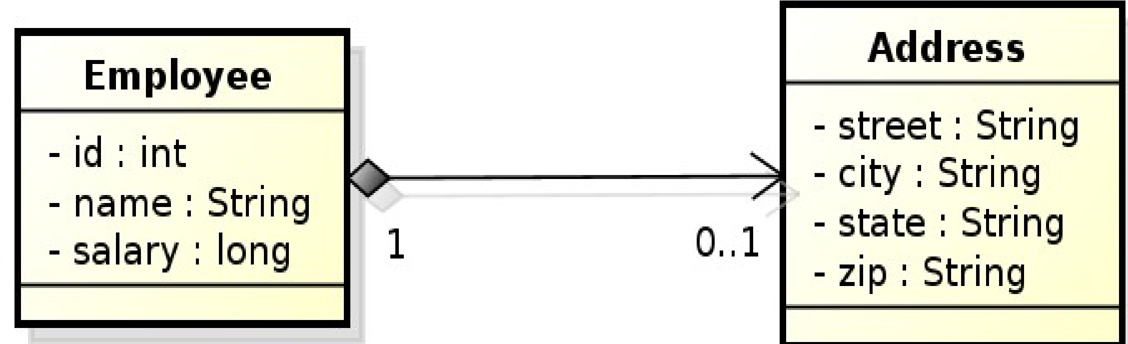
```
@Entity
public class Employee {

    @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<Phone> 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



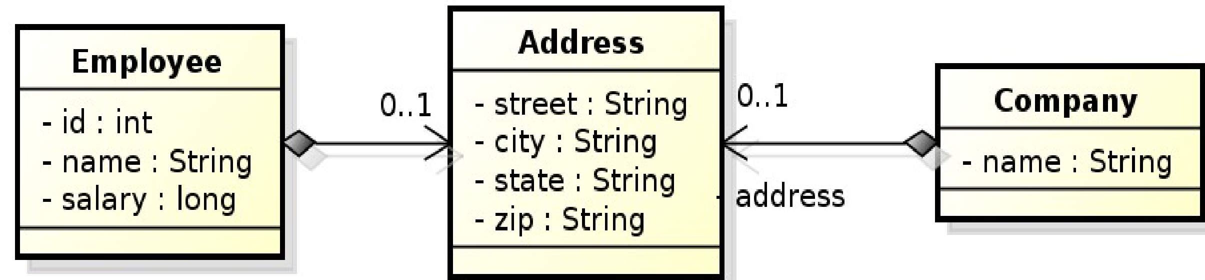

```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
   private String street;
   private String city;
   private String state;
   @Column(name="ZIP_CODE")
   private String zip;
}
```

```
@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

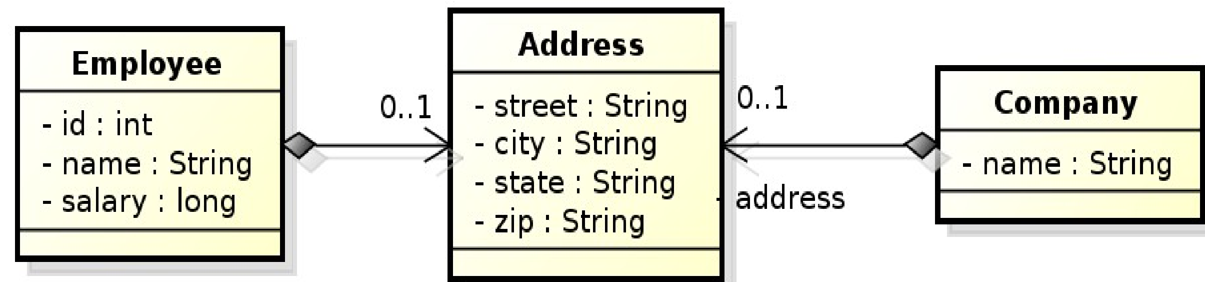



```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
   private String street;
   private String city;
   private String state;
   @Column(name="ZIP_CODE")
   private String zip;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE

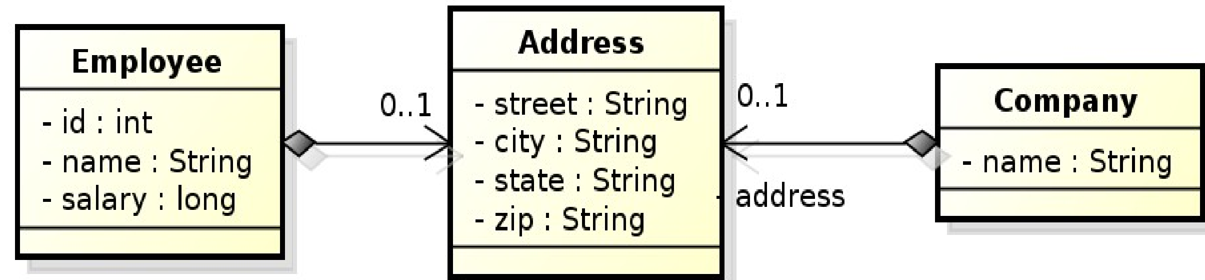



```
@Entity
public class Company {
    @Id private String name;
    @Embedded
    private Address address;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE



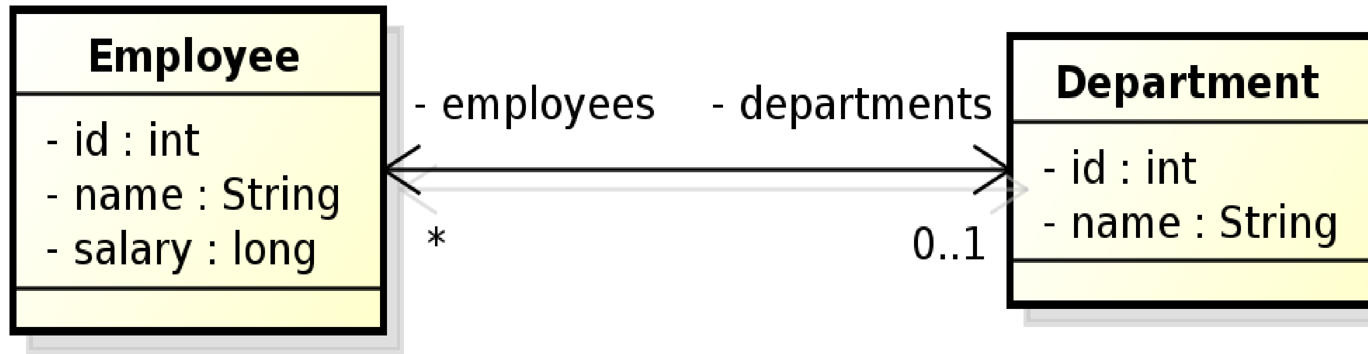

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name=“state“, column=@Column(name=“PROVINCE“)),
        @AttributeOverride(name=“zip“, column=@Column(name=“POSTAL_CODE“))
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

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);
emp.persist(addr);
emp.persist(emp);
```

Persisting bidirectional relationship




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

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

```
emp.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

laboratory Gerstner

ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

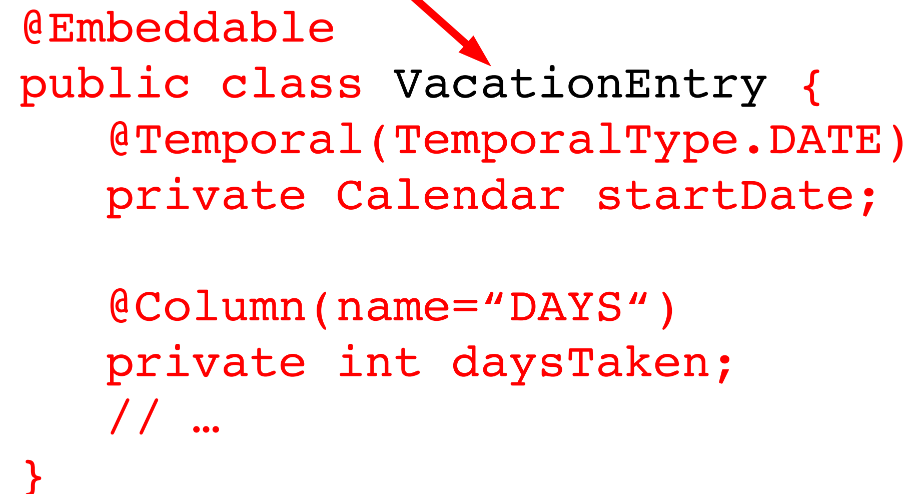
Collection Mapping

- 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
  - Set
  - List
  - Map

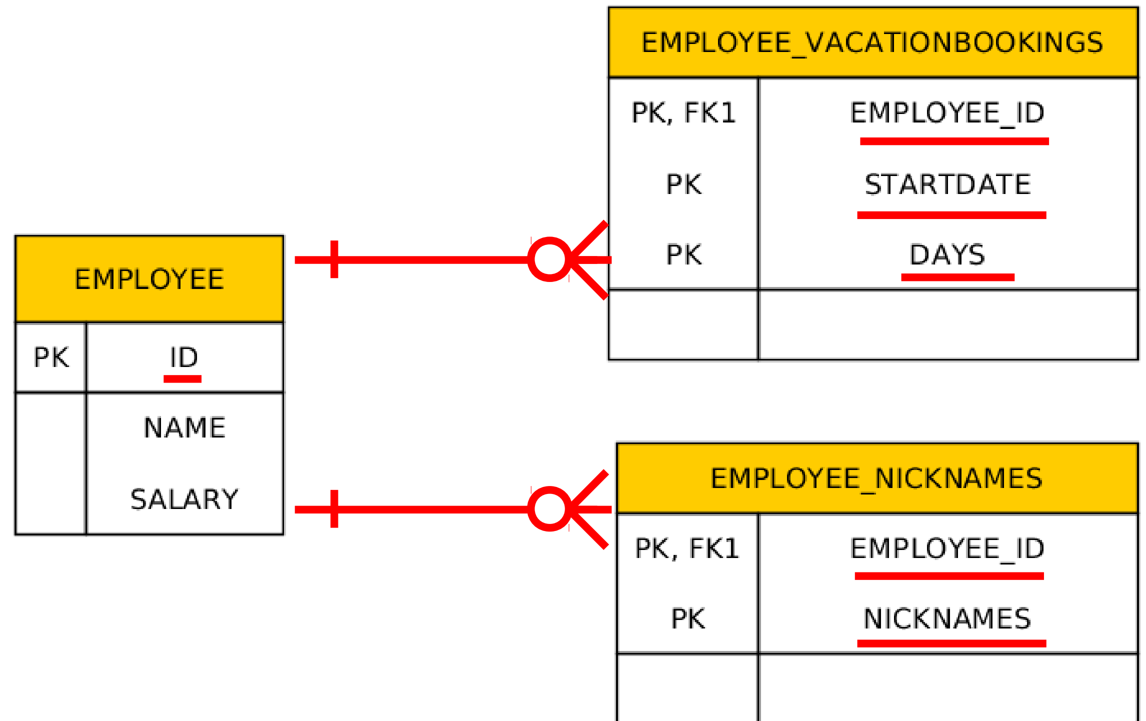
Collection Mapping

```
@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> nickNames;
    // …
}
```



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> nickNames;
   // …
}
```



Collection Mapping

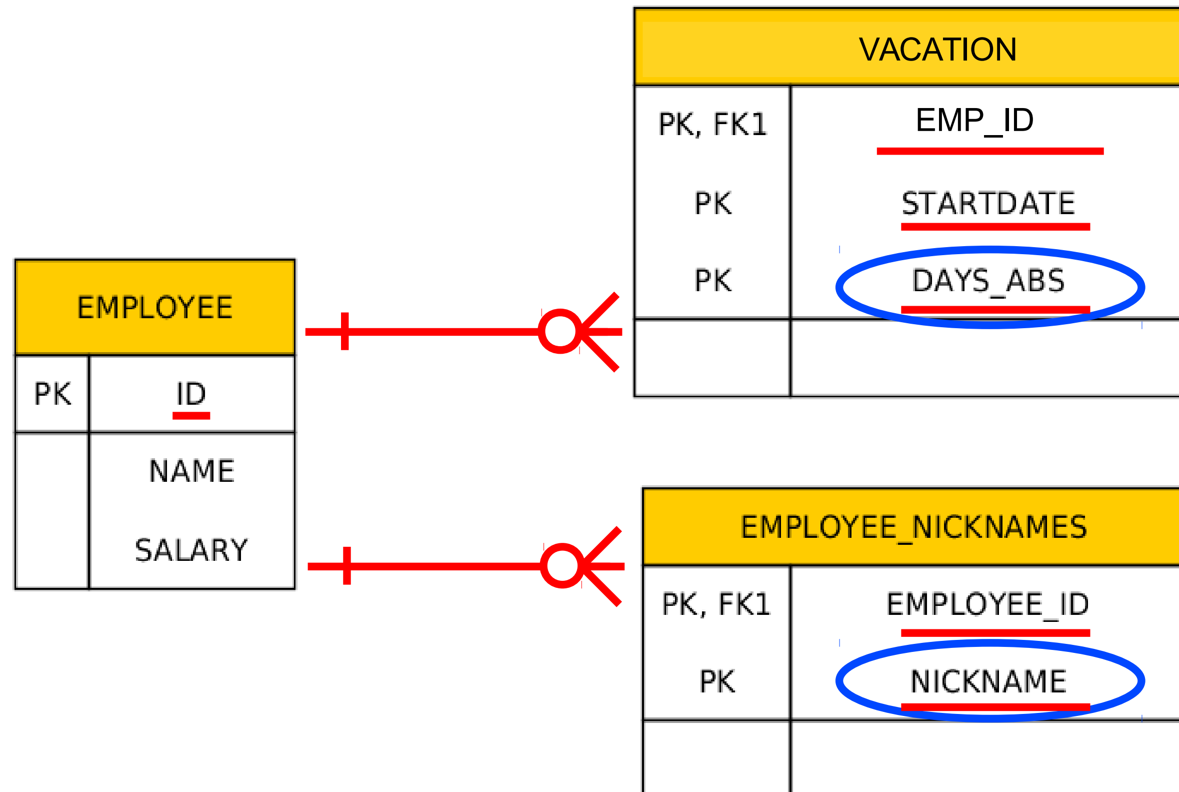
```
@Entity
public class Employee {
   @Id private int id;
   private String name;
   private long salary;
   // …
   @ElementCollection(targetClass=VacationEntry.class);
   @CollectionTable(
      name="VACATION",
      joinColumn=@JoinColumns(name="EMP_ID");
   @AttributeOverride(name="daysTaken",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

Interfaces:

- Collection
- Set
- List
- Map

may be used for mapping purposes.

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

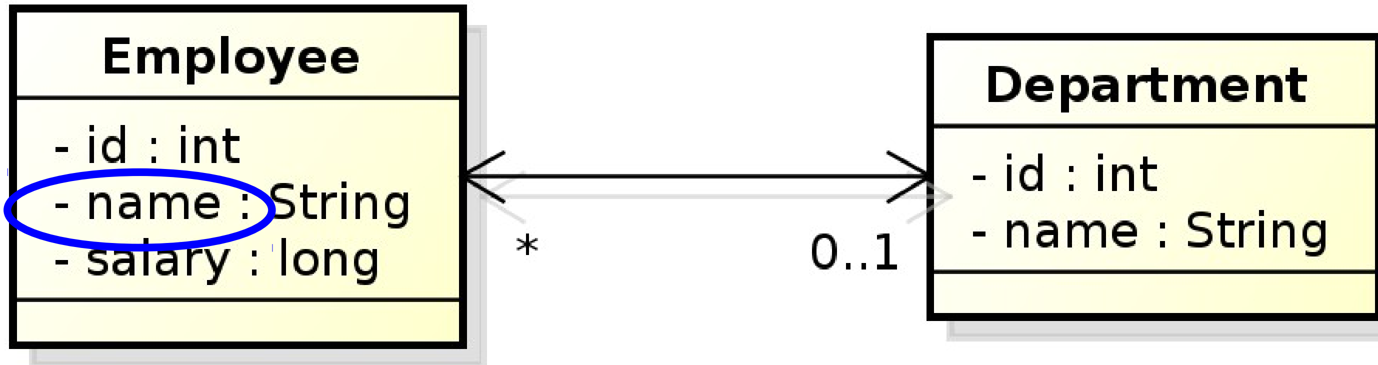
As soon as such an Entity becomes **managed** (by calling em.persist(...)), we can expect to get an instance of the respective interface, not an instance of that particular implementation class when we get it back (em.find(..)) to the persistence context. The reason is that the JPA provider may replace 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

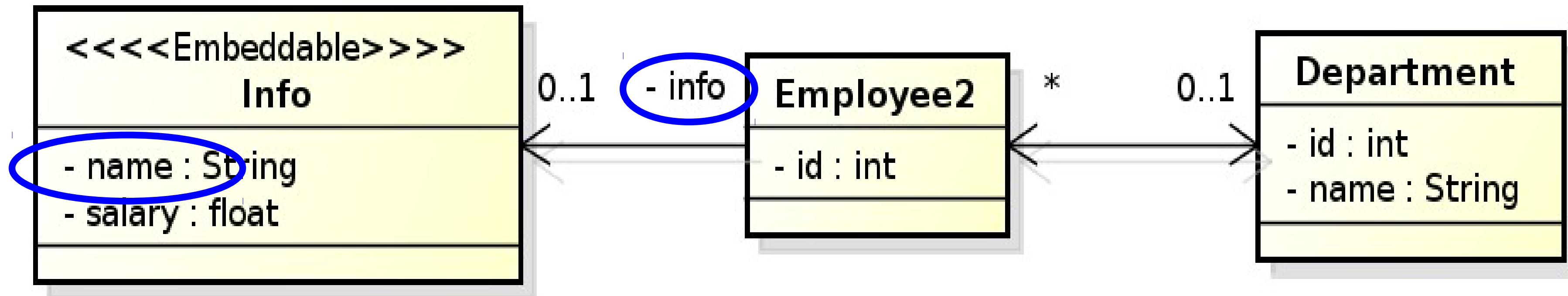
- Persistently ordered lists
  the ordering is persisted by means of an additional database column(s)
  typical example – ordering = the order in which the entities were persisted

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



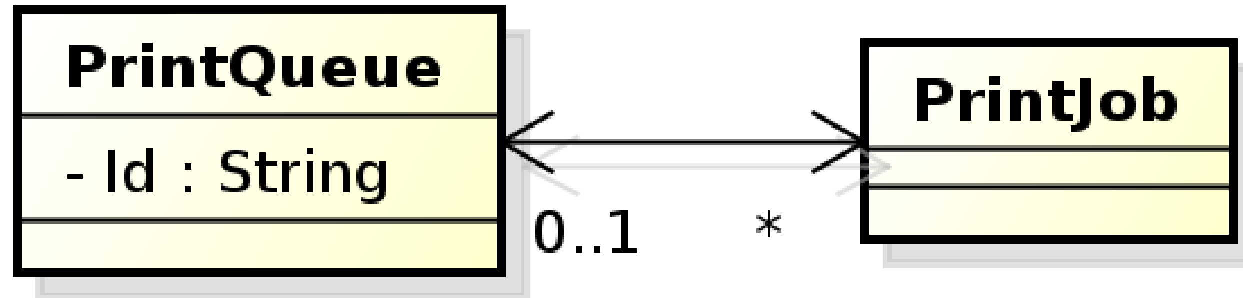

```
@Entity
public class Department {
    // …
    @OneToMany(mappedBy="department")
    @OrderBy("name ASC")
    private List<Employee> employees;
    // …
}
```

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



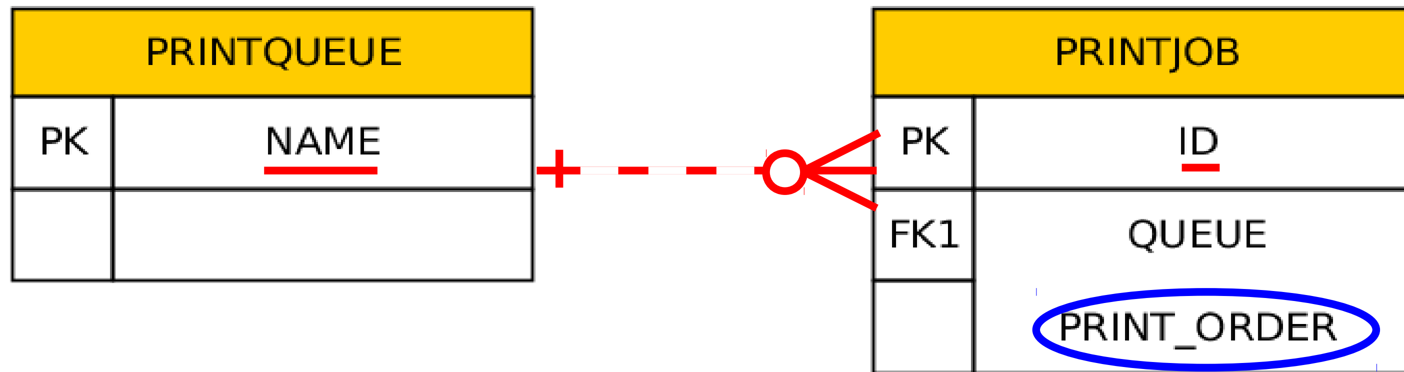

```
@Entity
public class Department {
    // …
    @OneToMany(mappedBy="department")
    @OrderBy("info.name ASC")
    private List<Employee2> employees;
    // …
}
```

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:

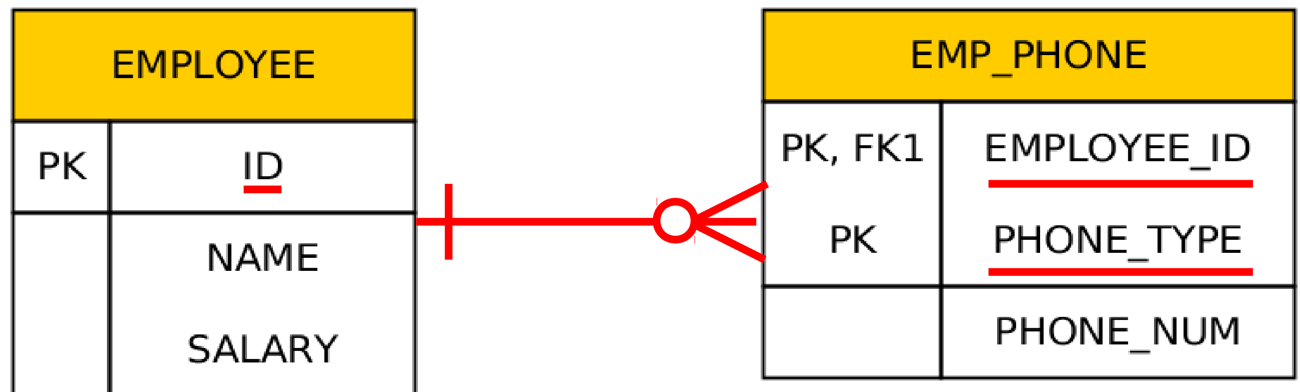
- Basic types (stored directly in the table being referred to)
  - Target entity table
  - Join table
  - Collection table
- Embeddable types (- “ -)
- Entities (only foreign key is stored in the table)

Values:

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

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

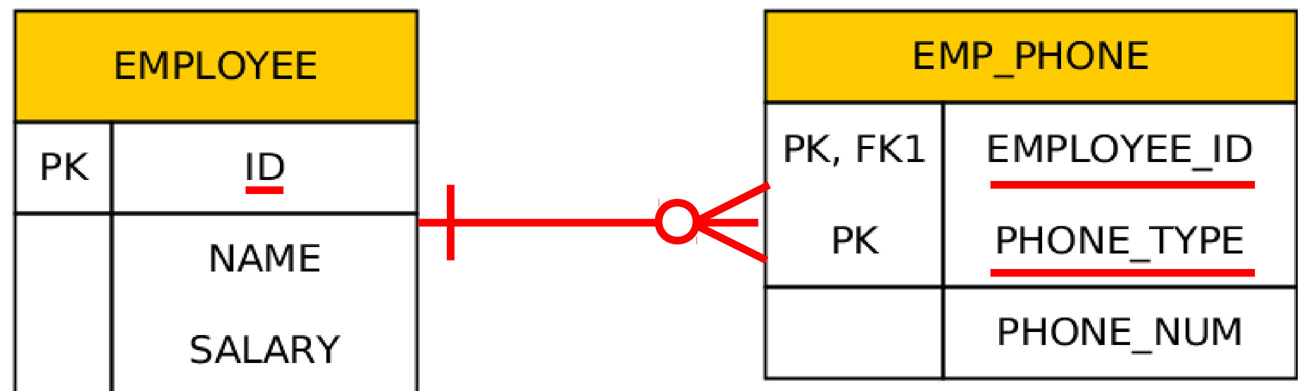
```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @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;
    // …
}
```

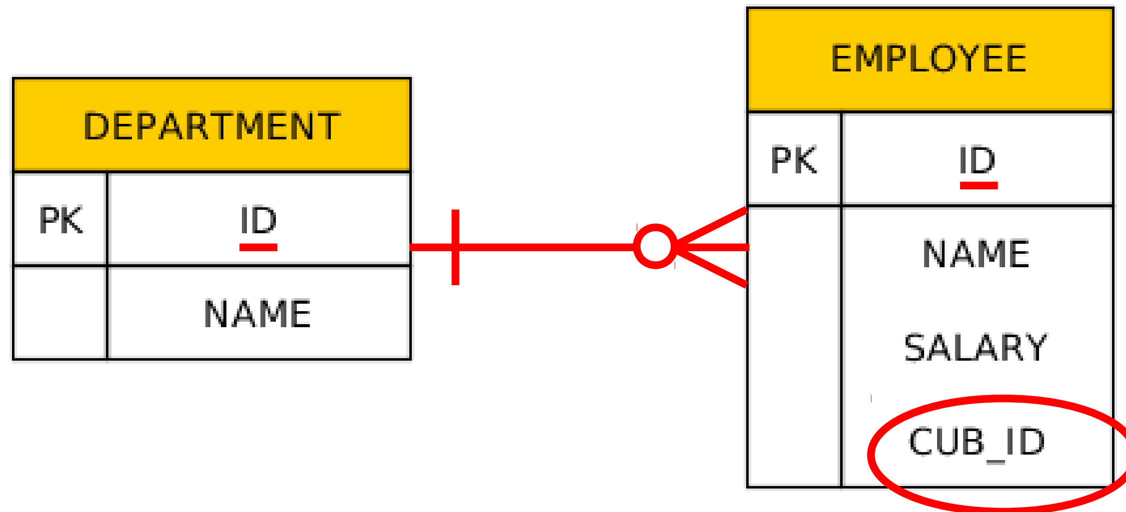
```
Public enum PhoneType {
        Home,
        Mobile,
        Work
}
```



Collection Mapping – Maps

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

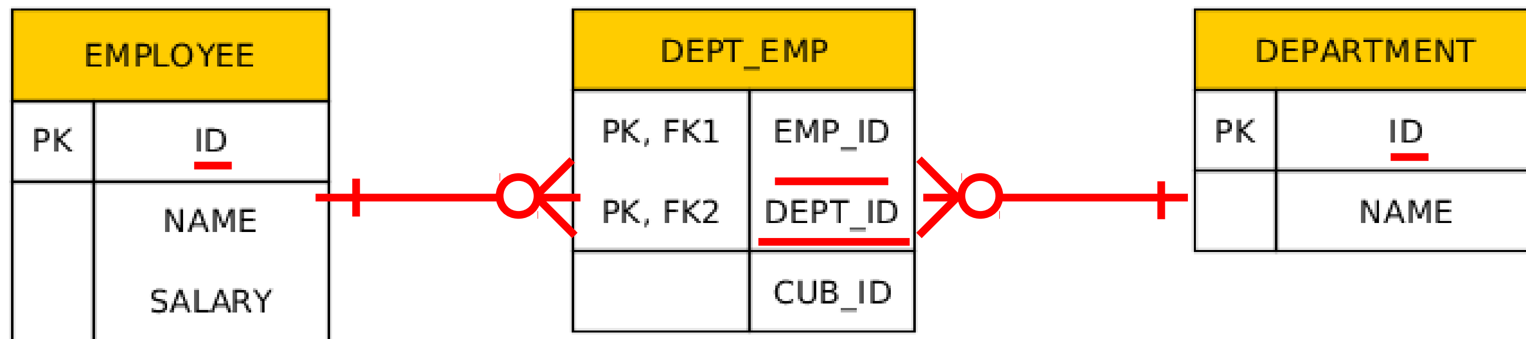
```
@Entity
public class Department {
   @Id private int id;
   private String name;

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



Collection Mapping – Maps

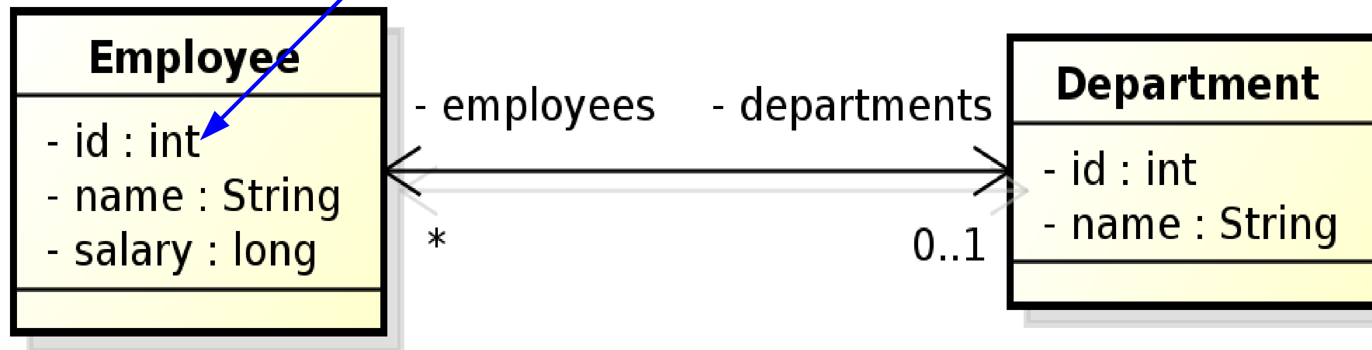
(keying by basic type – N:M relationship using a Map with String key)

```
@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;
    // …
}
```



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;
    // ...
}
```

laboratory Gerstner

ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

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 EmployeeId(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();
```

Optionality

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

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

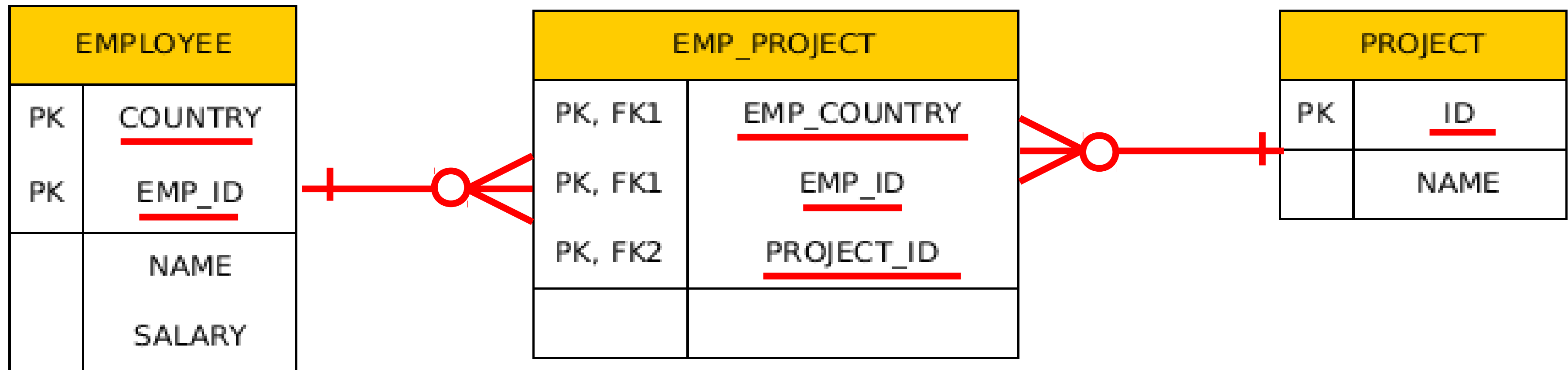
Optionality (parciality) can be used only for @ManyTo**One** and @OneTo**One** relations making the „1“ side of the cardinality „0...1.“

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;
    // …
}
```

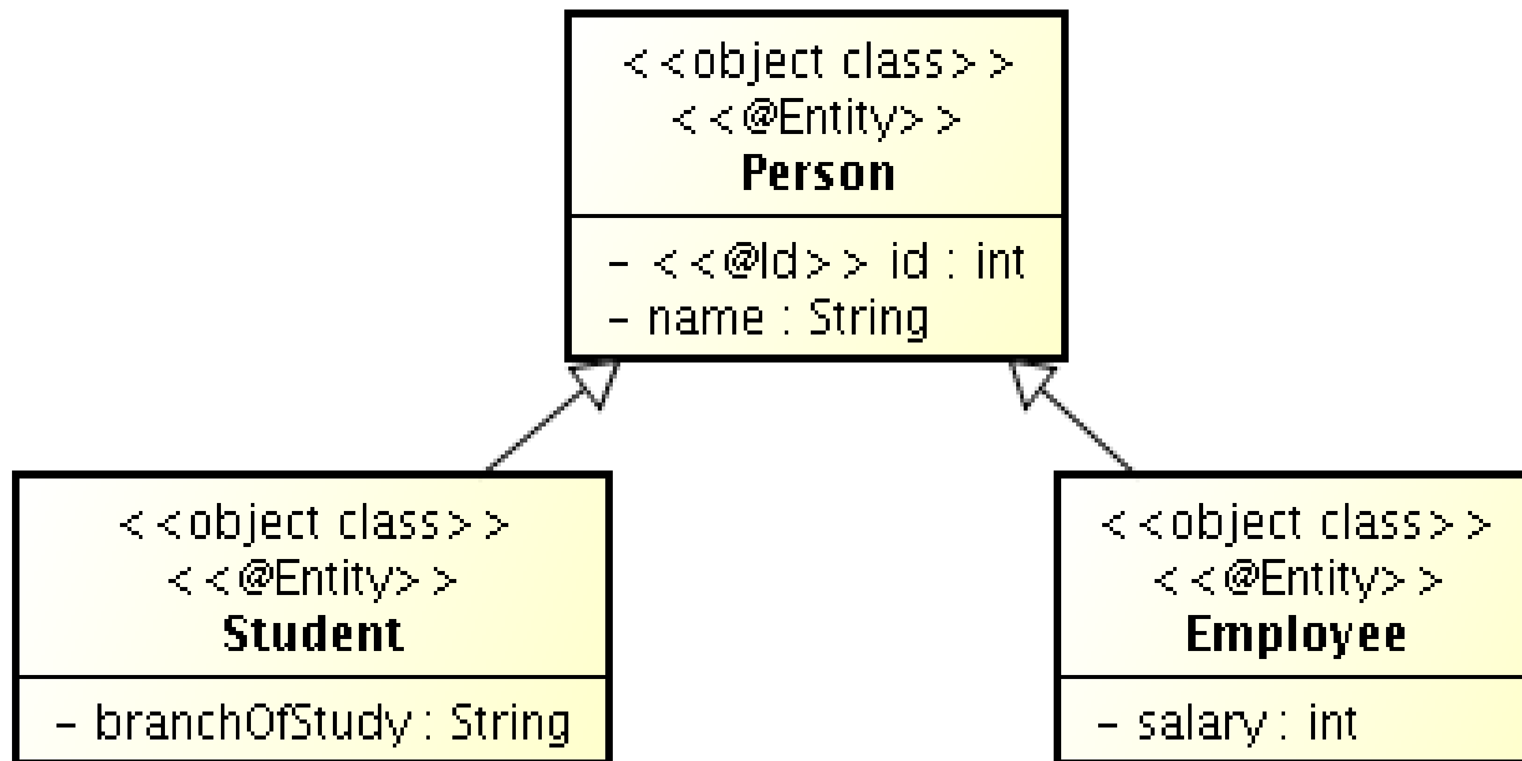
Compound Join Columns




```
@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;
}
```

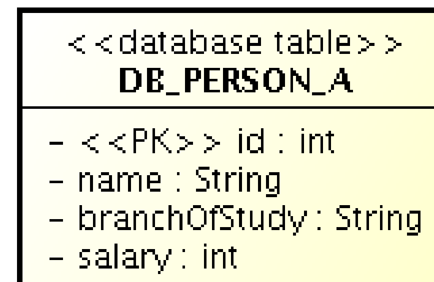
Inheritance

- How to map inheritance into RDBMS ?

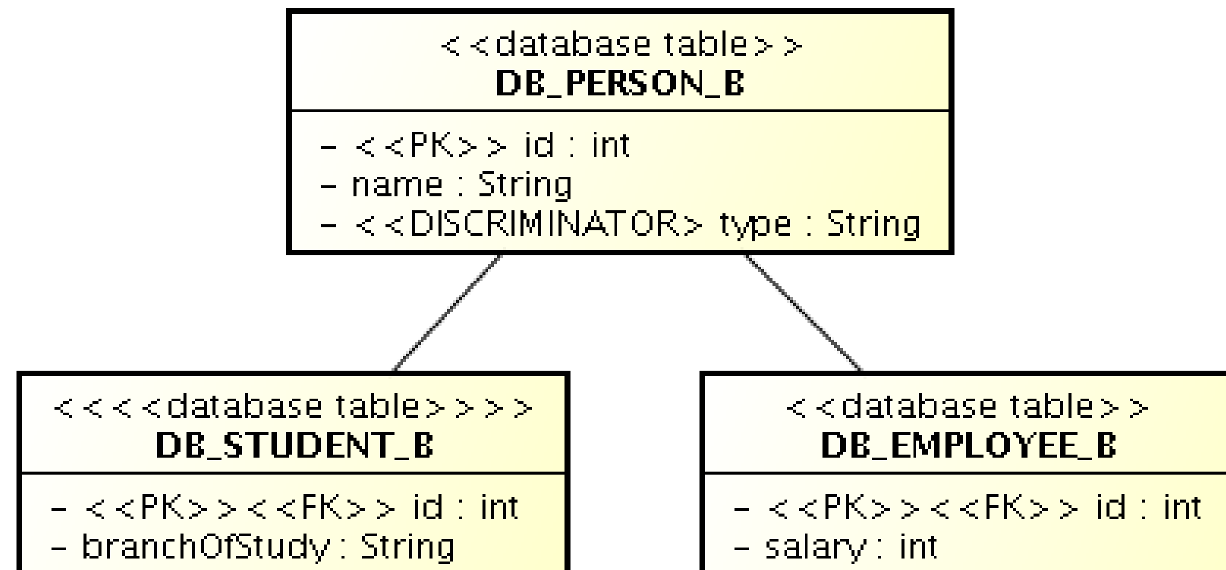


Strategies for inheritance mapping

- Single table

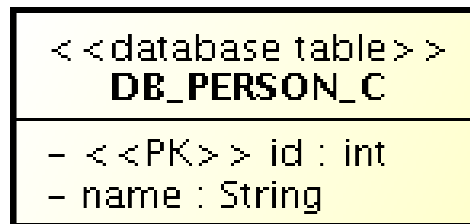


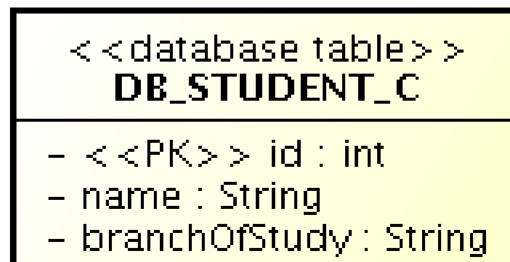
- Joined

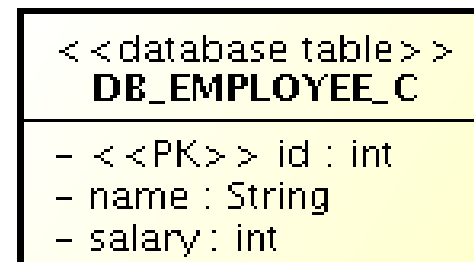


Strategies for inheritance mapping

- Table-per-concrete-class







Inheritance mapping
single-table strategy

```
@Entity
@Table(name="DB_PERSON_A")
@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

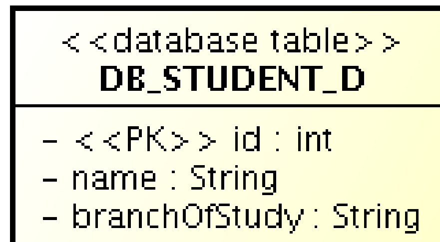
```
@Entity
@Table(name="DB_PERSON_B")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminationColumn(name="EMP_TYPE",
                      discriminatorType=discriminatorType.INTEGER)
public abstract class Person { …}

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

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

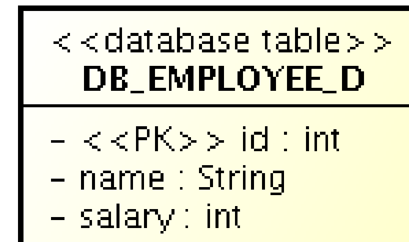
Inheritance mapping
table-per-concrete-class strategy

```
@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.

Queries

- JPQL (Java Persistence Query Language)
- Native queries (SQL)

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 are not defined 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

```
public class Query {
   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

```
static final 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 for 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

```
@SqlResultSetMapping(name="OrderResults",
        entities={
            @EntityResult(entityClass=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")}
    )
```

```
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");

List<Object[]> results = q.getResultList();

results.stream().forEach((record) -> {
    Order order = (Order)record[0];
    String itemName = (String)record[1];
    /...
});
```