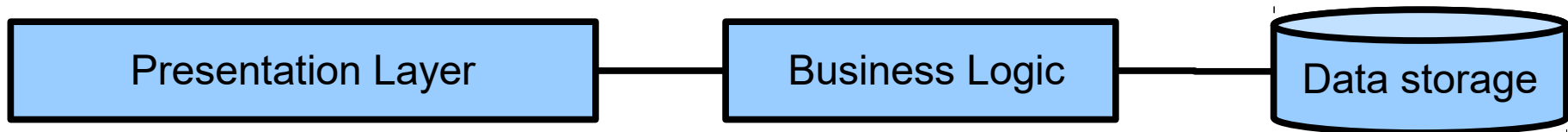


ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

What is Object-relational mapping ?

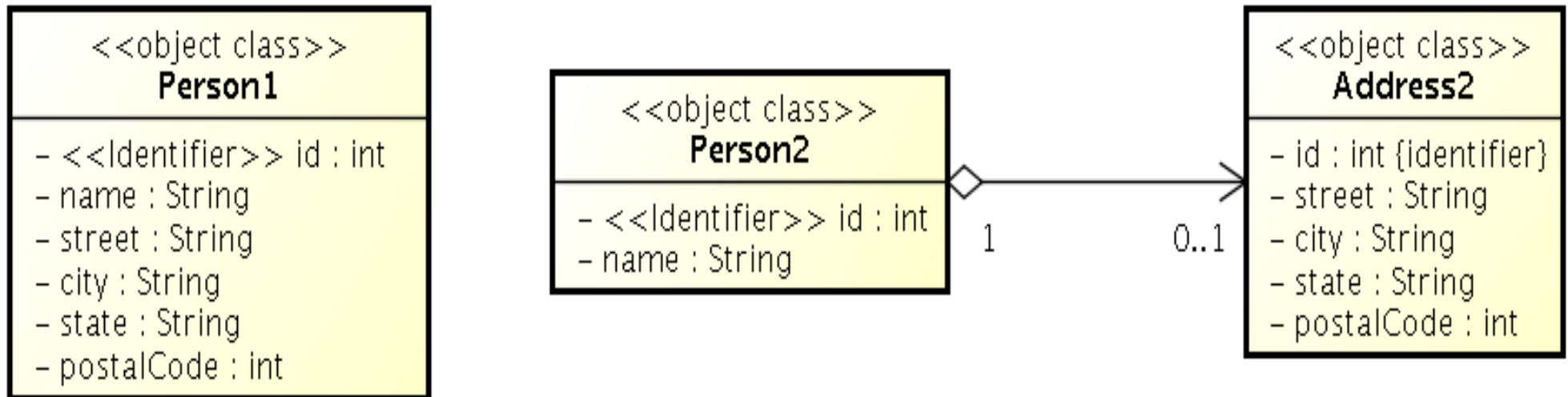
- a typical information system architecture:



- 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 ?

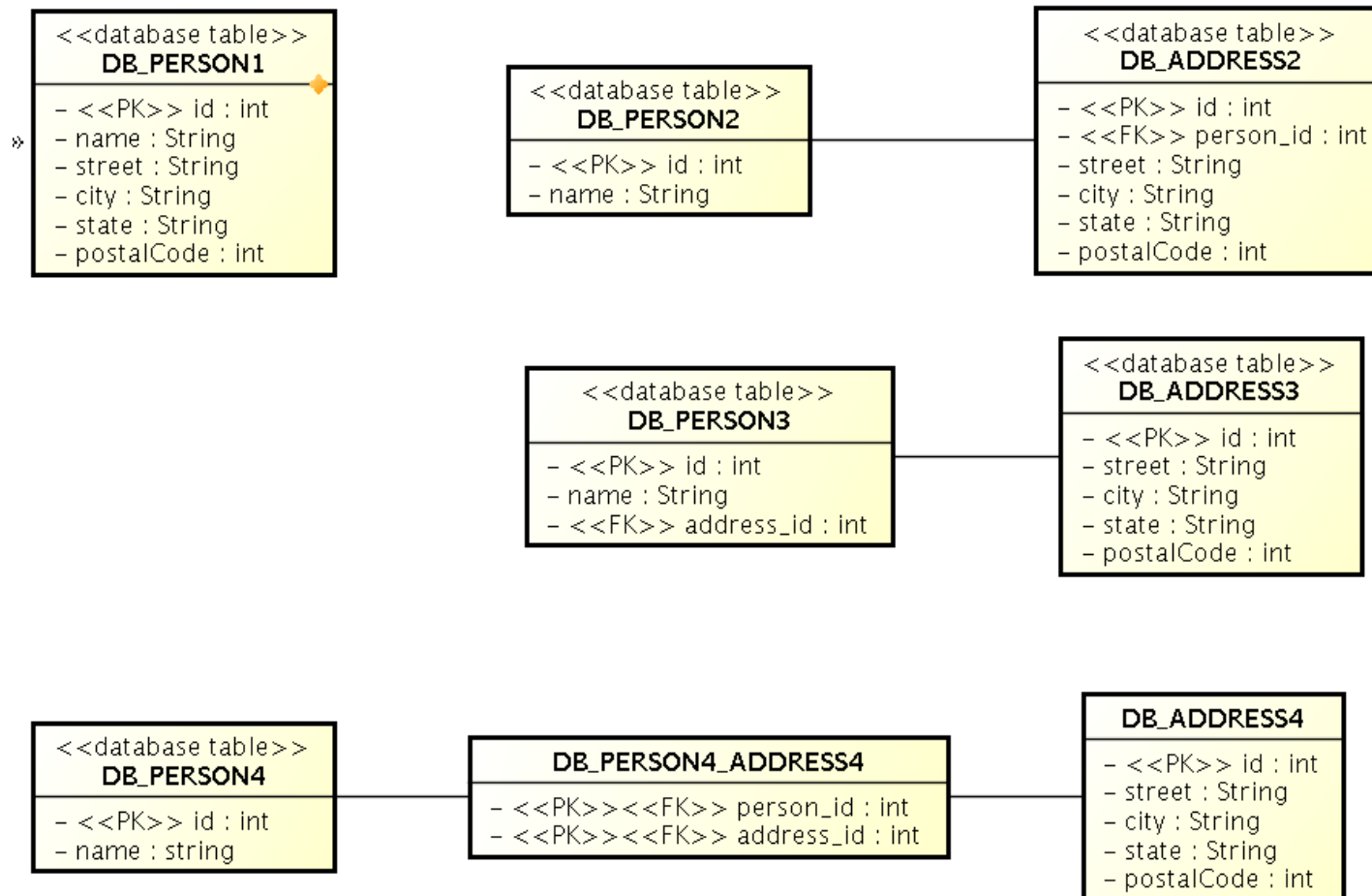
Example – object model

- When would You stick to one of these options ?



Example – database

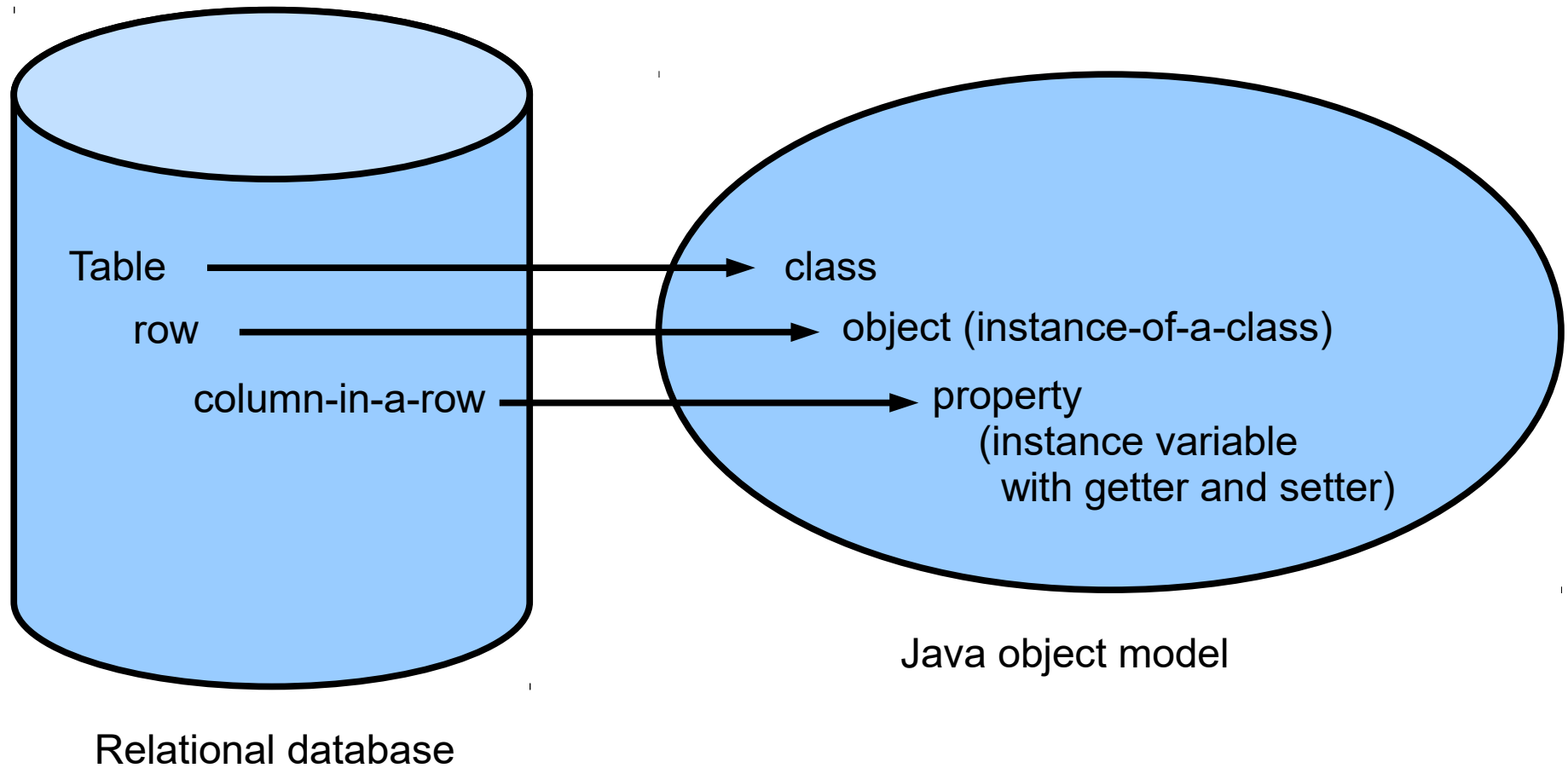
- ... and how to model it in SQL ?



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
```

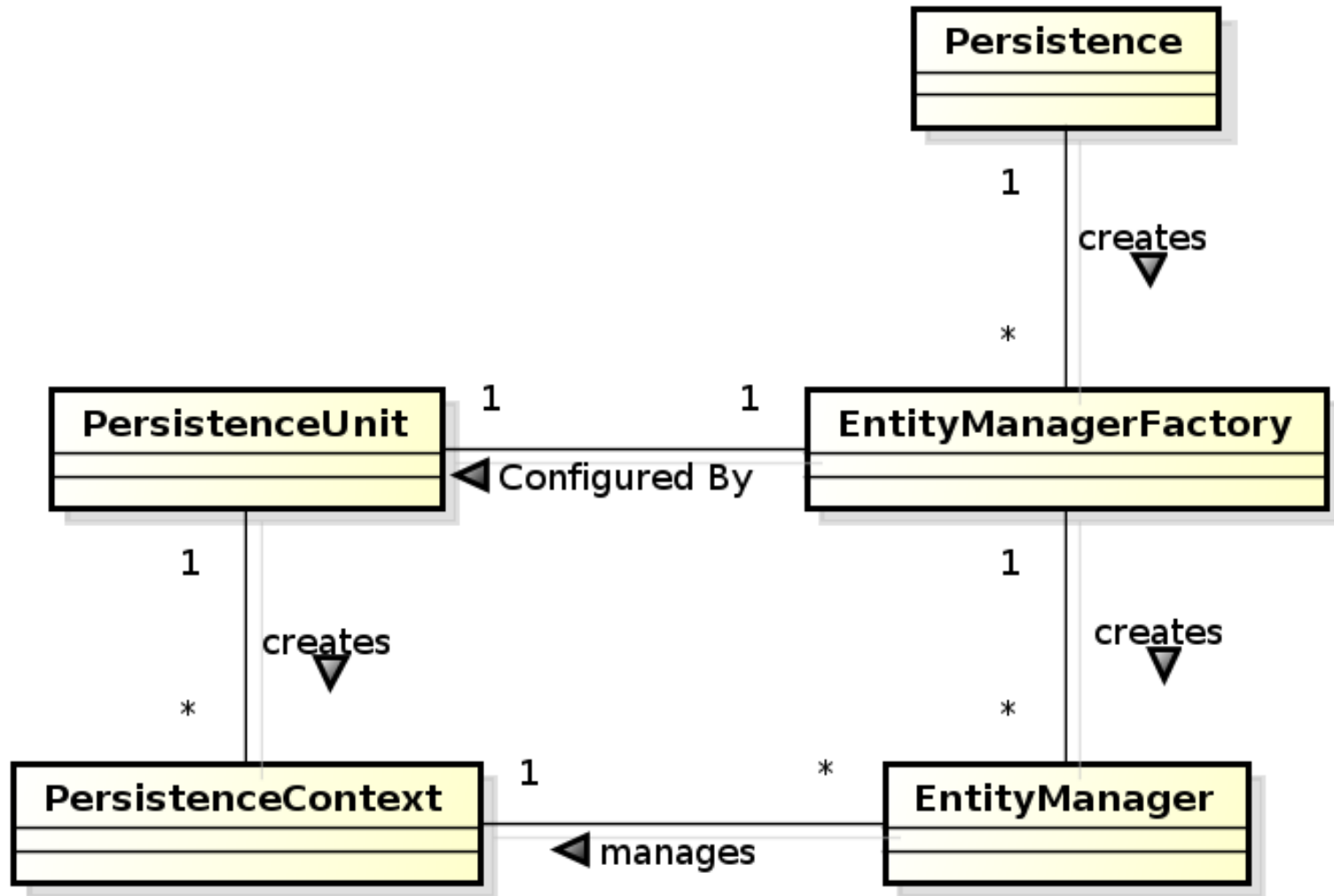
```
    private Integer id;
```

```
    private String name;
```

```
    // setters + getters
```

```
}
```


JPA2.0 – Basic concepts

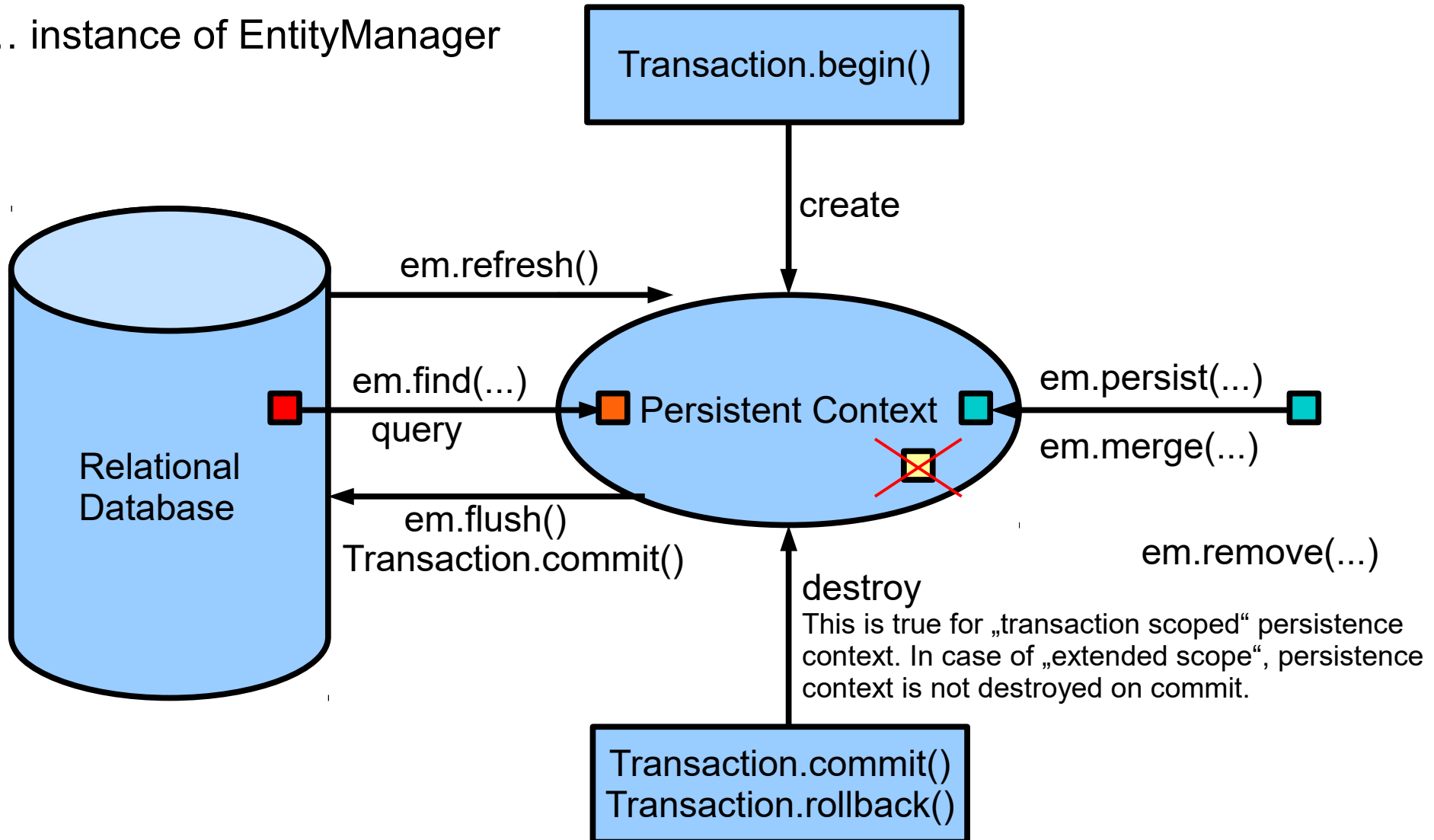


JPA 2.0 - Basics

- 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)

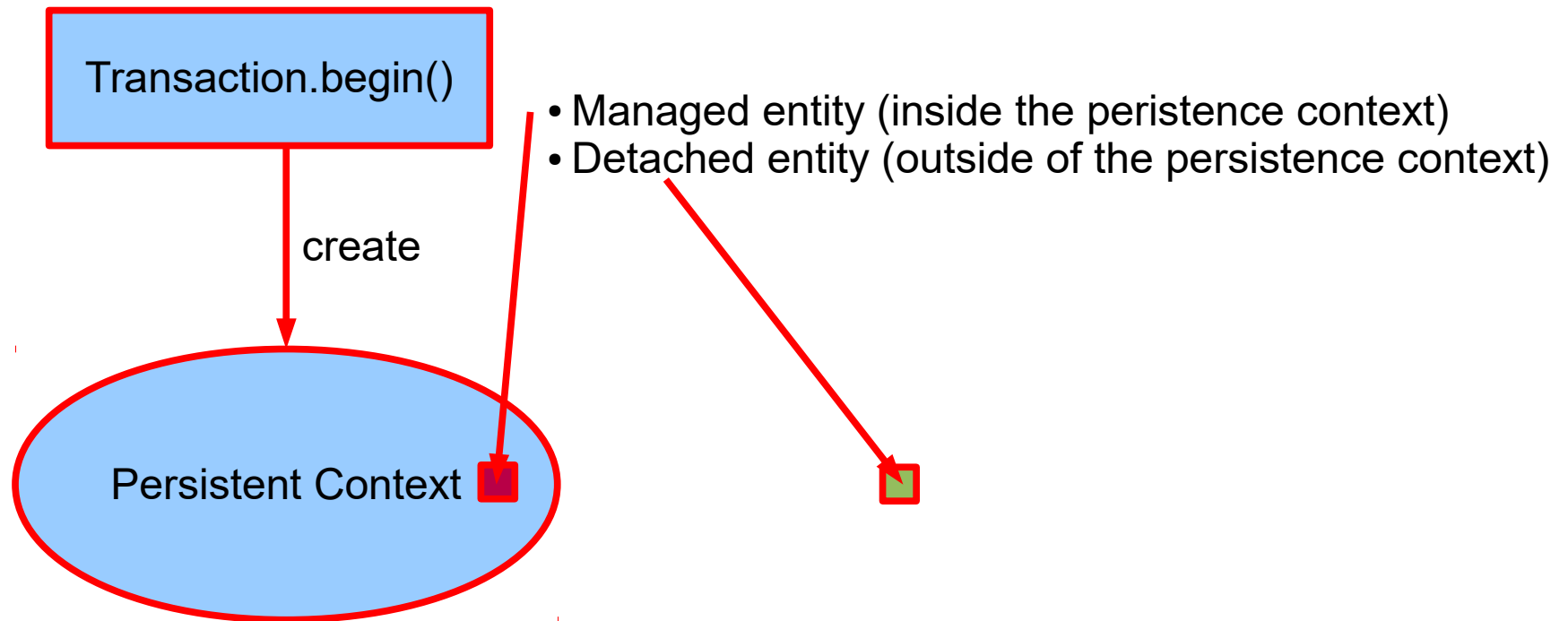
JPA 2.0 – Persistence Context

em ... instance of EntityManager



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 understood as DPO (managed) or DTO (detached).
- Selected operations on EM (CRUD) :
 - **Create** : em.persist(Object o)
 - **Read** : em.find(Object id), em.refresh(Object o)
 - **Update** : em.merge(Object o)
 - **Delete** : 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 setId(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 overridden 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(String 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

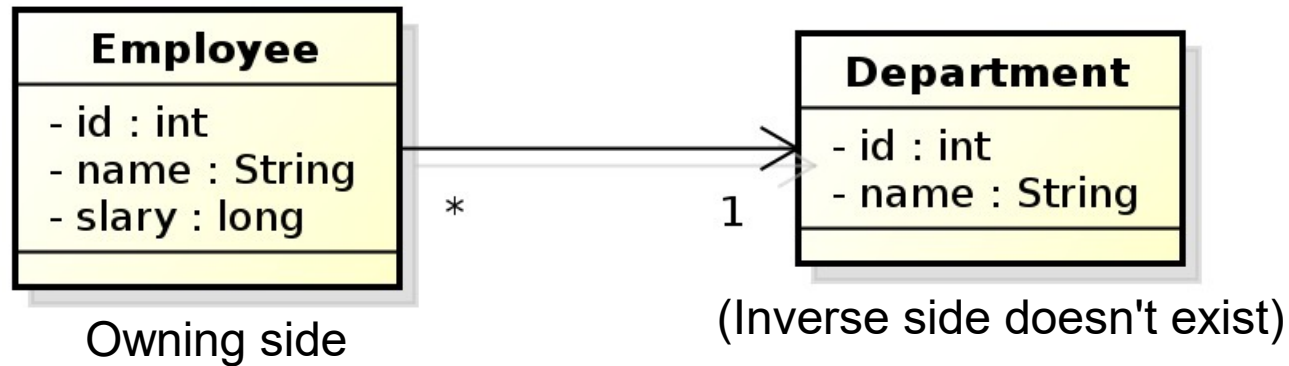
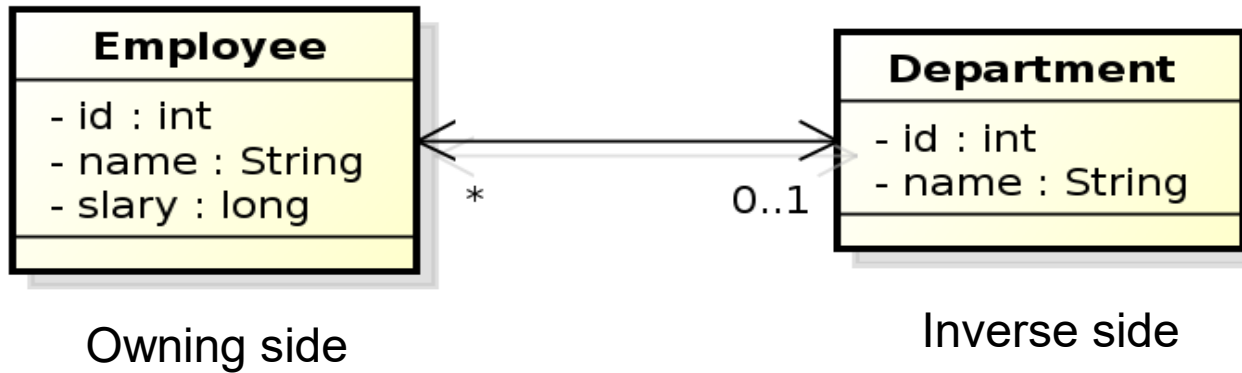
Strategies

- 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;
```

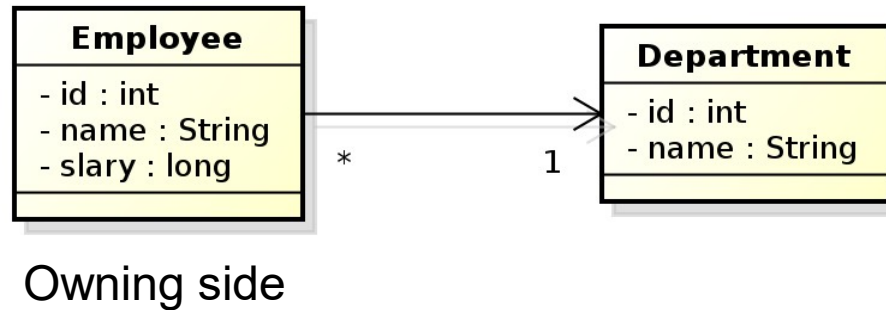

ORM – Relationships



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)

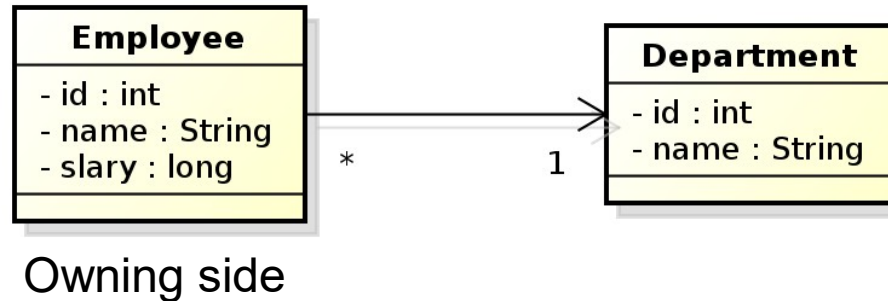
Unidirectional many-to-one relationship



```
@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



@Entity

```
public class Employee {
```

```
    @Id private int id;
```

```
    private String name;
```

```
    @ManyToOne
```

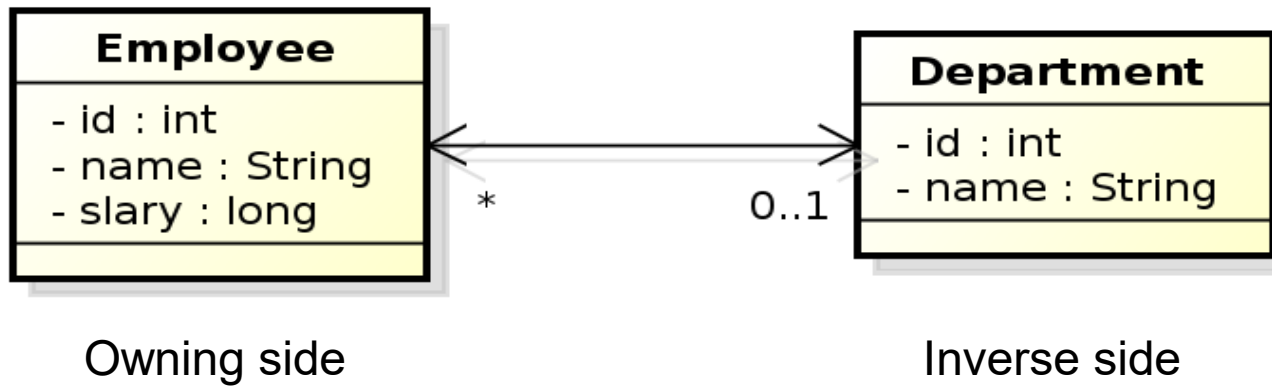
```
    @JoinColumn(name="DEPT_ID")
```

```
    private Department department;
```

```
}
```

In this case, the foreign key is defined as the @JoinColumn annotation.

Bidirectional many-to-one relationship



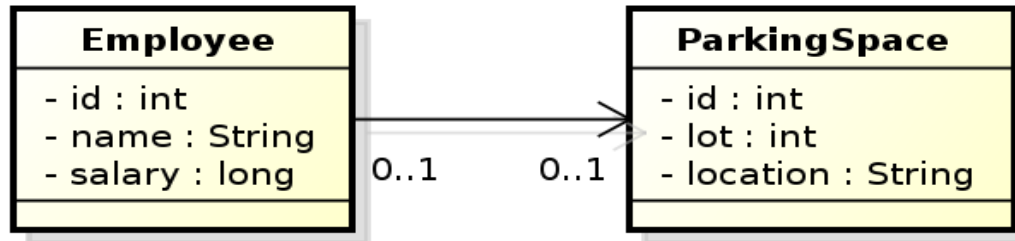
```
@Entity
public class Employee {

    @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;
```

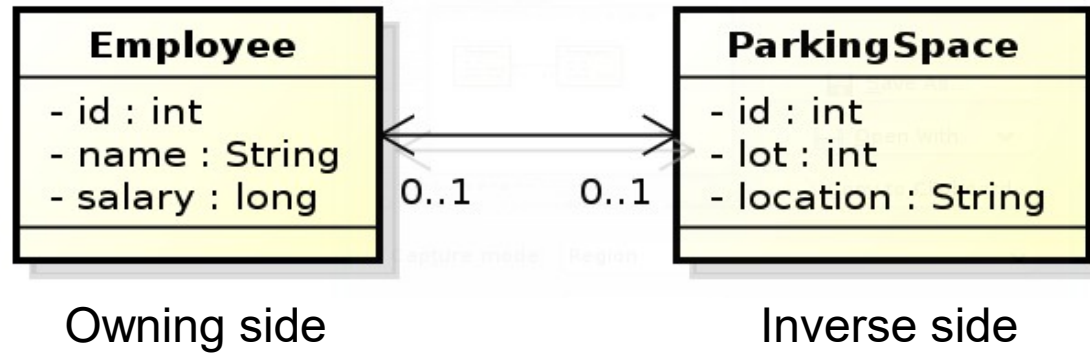
```
    @OneToOne
```

```
    @JoinColumn(name="PSPACE_ID")
```

```
    private ParkingSpace parkingSpace;
```

```
}
```

Bidirectional one-to-one relationship



```
@Entity
public class Employee {

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

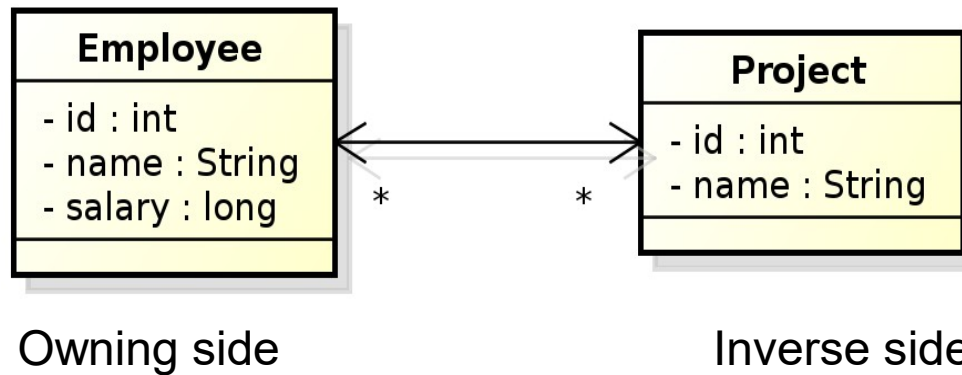
}
```

```
@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



```
@Entity
public class Employee {
```

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

```
}
```

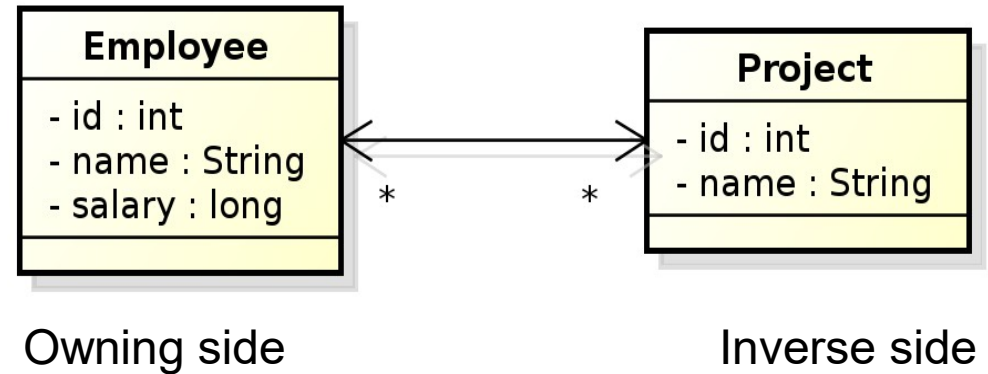
```
@Entity
public class Project {
```

```
    @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



```
@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;
```

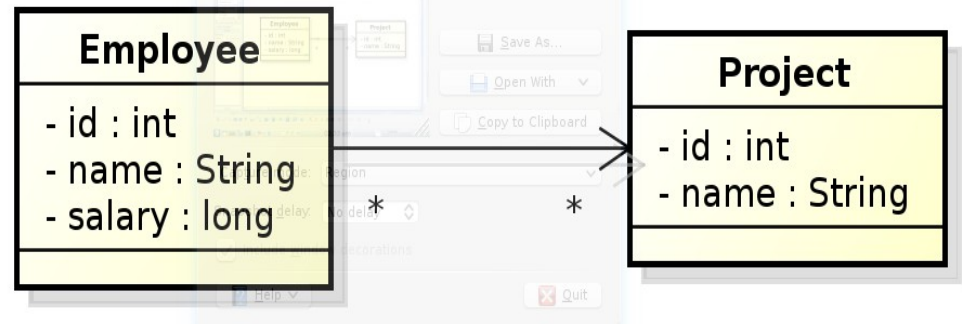
```
}
```

```
@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;

}
```

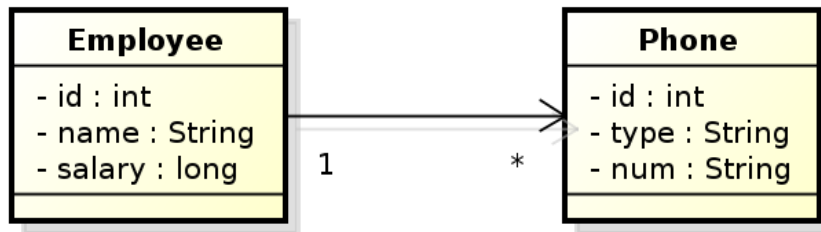
```
@Entity
public class Project {

    @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")
```

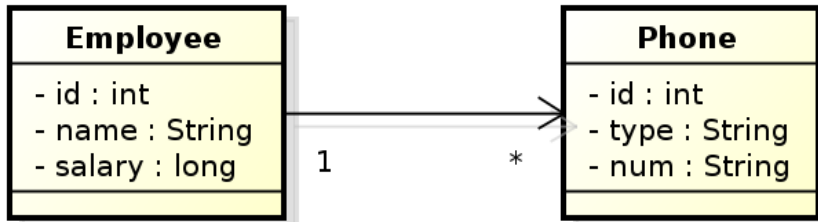
```
    private Collection<Phone> phones;
```

```
}
```

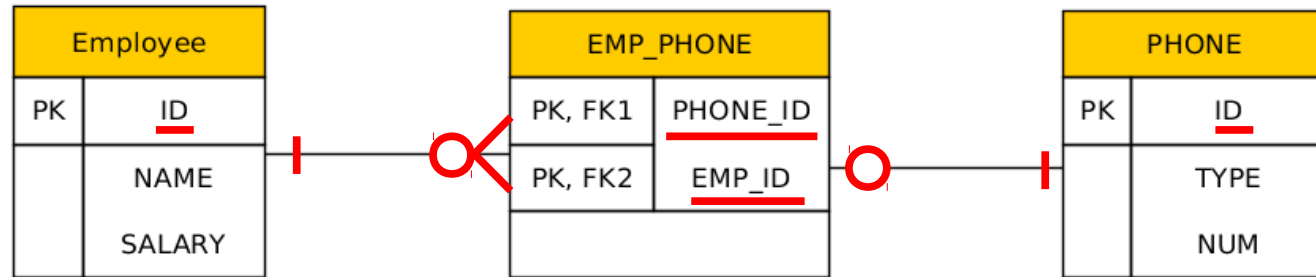
Not available prior to JPA 2.0

// 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

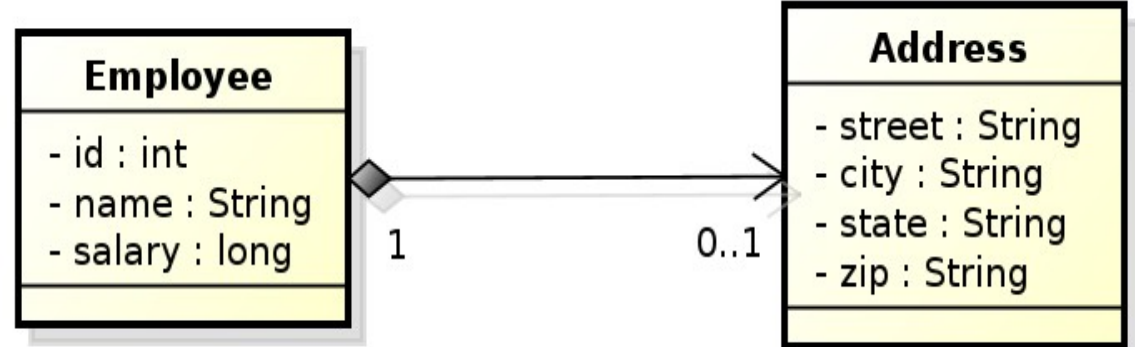
```
@Entity
public class Employee {

    @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



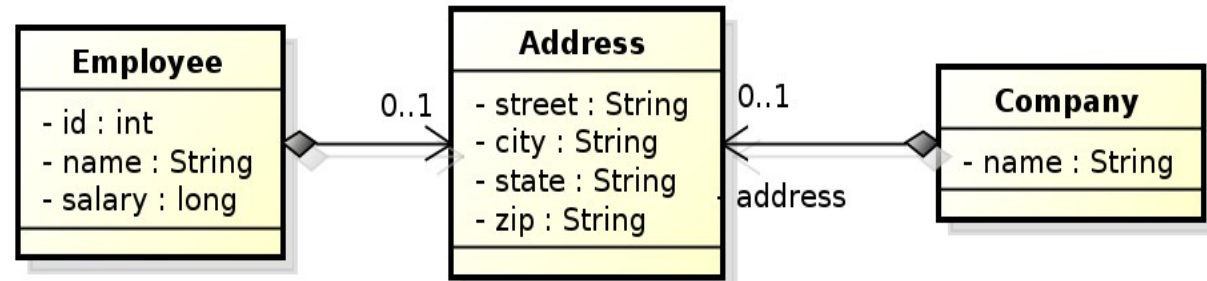
```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

```
@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

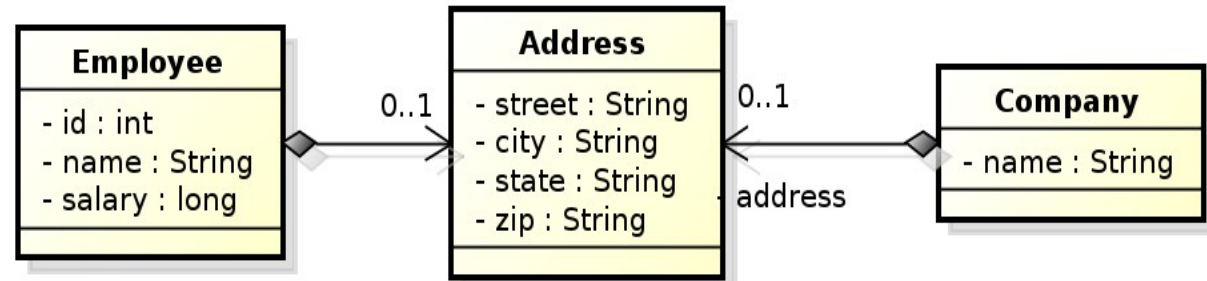


```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE



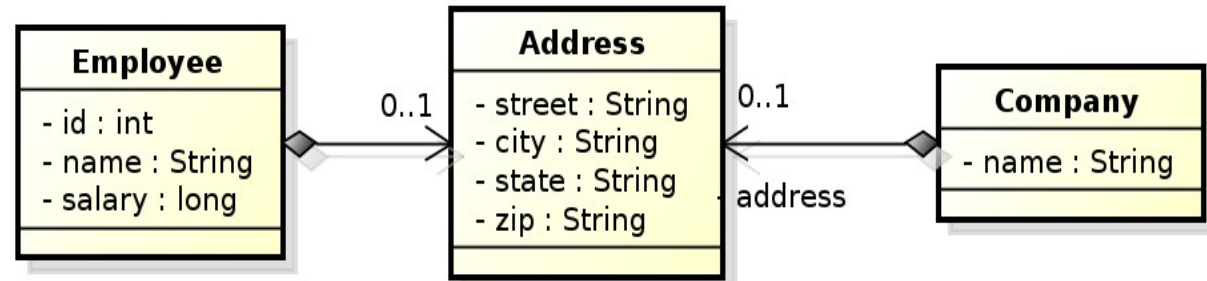
@Entity

```
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

```

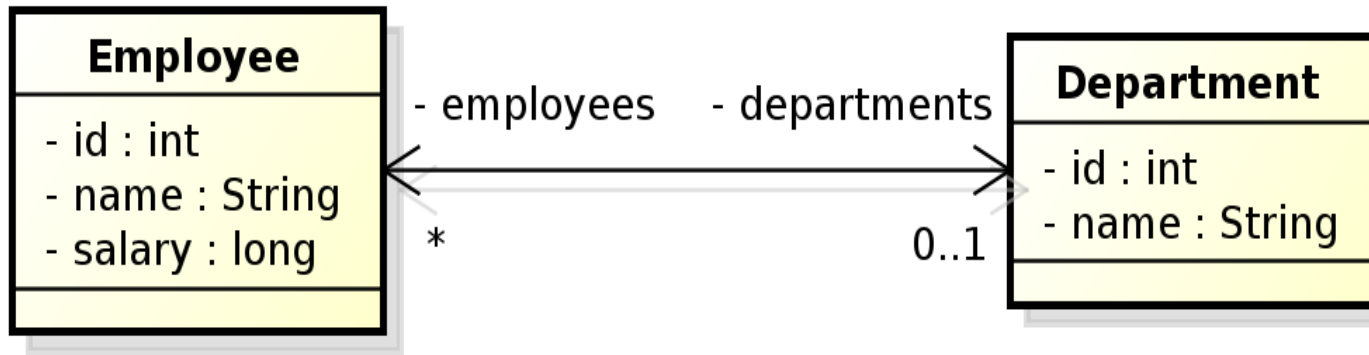
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
    })
    private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