Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

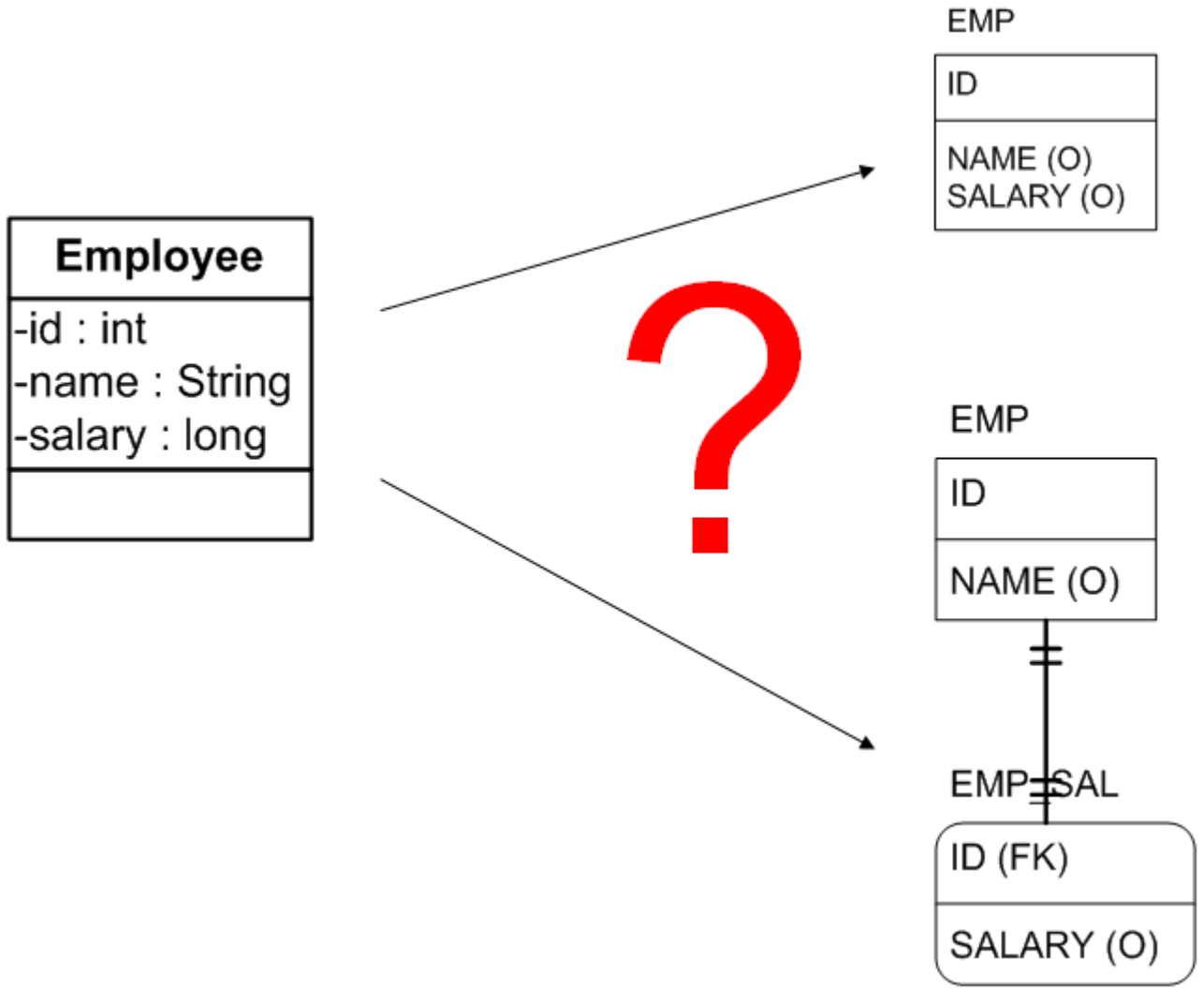
Java Persistence API

Simon Martinelli

CONTENT

1 INTRODUCTION

THE PROBLEM


Employee
-id : int
-name : String
-salary : long
EMP
ID
NAME (O)
SALARY (O)
?
EMP
ID
NAME (O)
EMP_SAL
ID (FK)
SALARY (O)

OBJECT-RELATIONAL IMPEDANCE MISMATCH

- **Structure**
  An object contains both data and behavior

- **Identity**
  An object has an identity independent of its state, while the identity of a record is determined by its data (primary key)

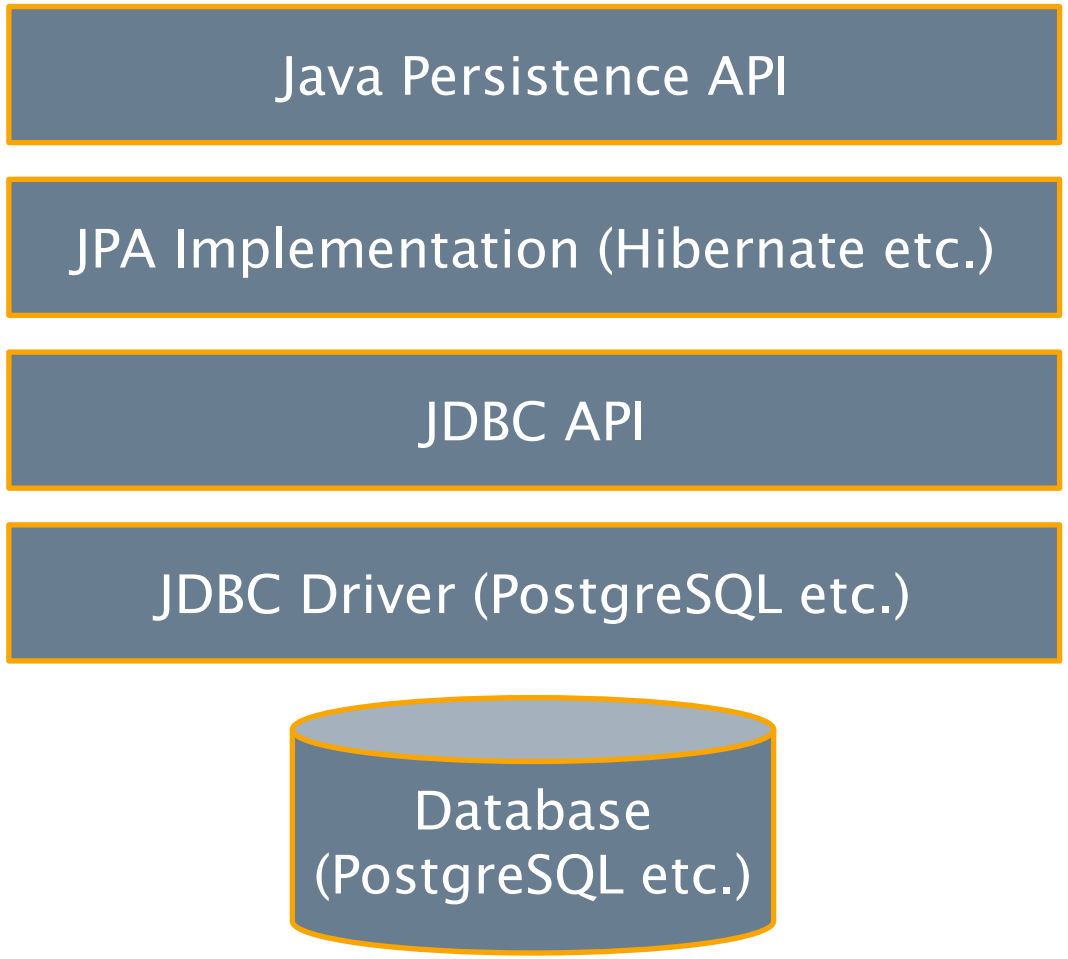
- **Data encapsulation**
  An object protects its data by limiting the way it can be changed

- **Transactionality**
  The data of a relational database is modified by transactions

MODERN PERSISTENCE APIS

- Work with ordinary Java classes for data (POJOs)
- Support
  - relationship mapping
  - transitive persistence
  - automatic dirty checking
  - lazy loading
- Minimize database roundtrips (join fetching)
- Generate SQL at runtime

TECHNOLOGY STACK



2 GETTING STARTED

ENTITY CLASS

- Class annotated with @Entity

- Requirements:
  - There is a field annotated as **primary key** (@Id)
  - The **standard constructor** must be present
  - The class must **not be final** and must not contain final methods
  - Fields must be **private** or **protected**

ENTITY EXAMPLE

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

  // getters/setters
}
```

ENTITY STATE

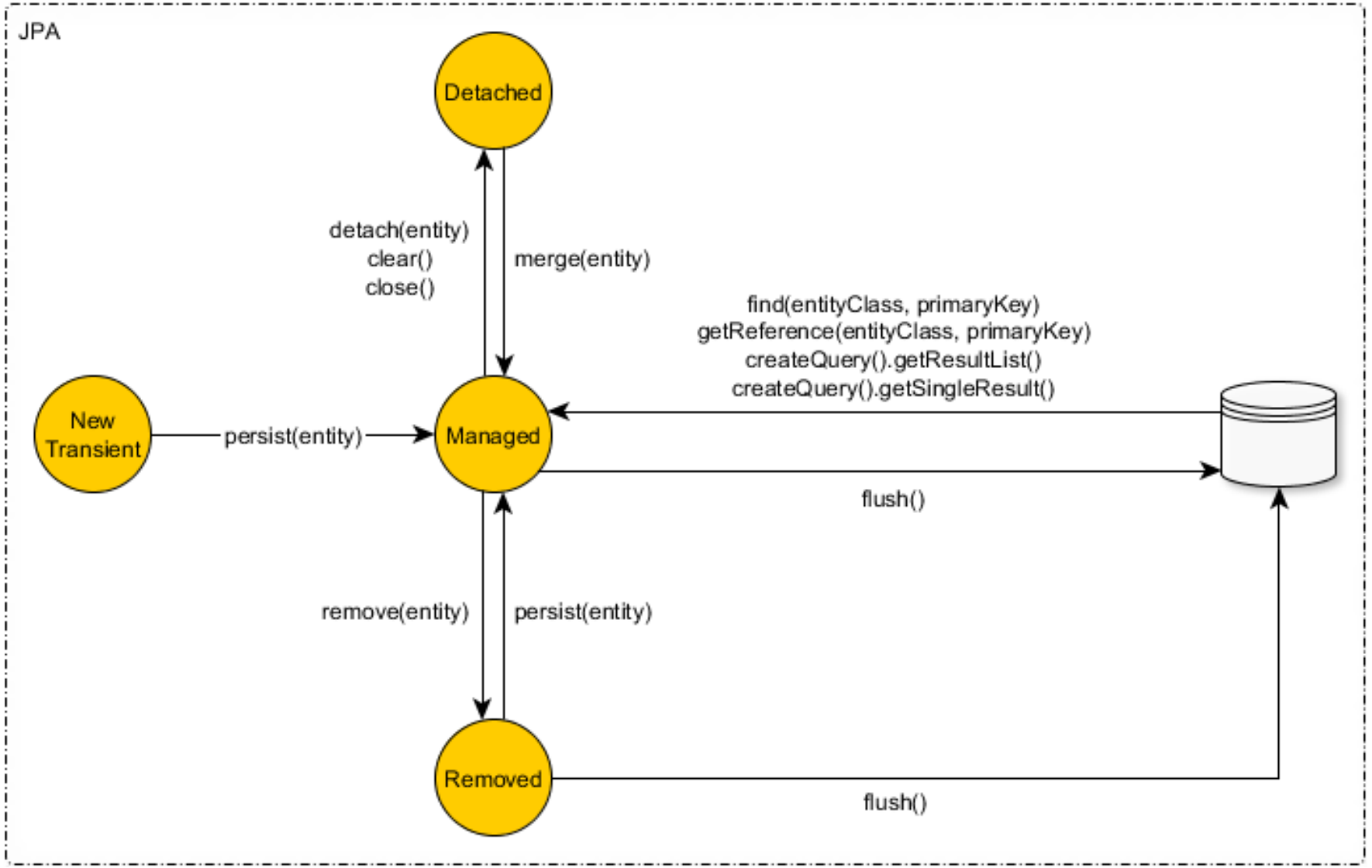
- **New**
  Object is newly created, has no connection with the database and no valid ID

- **Managed**
  The object has a record in the database, and changes are tracked automatically and synchronized with the database

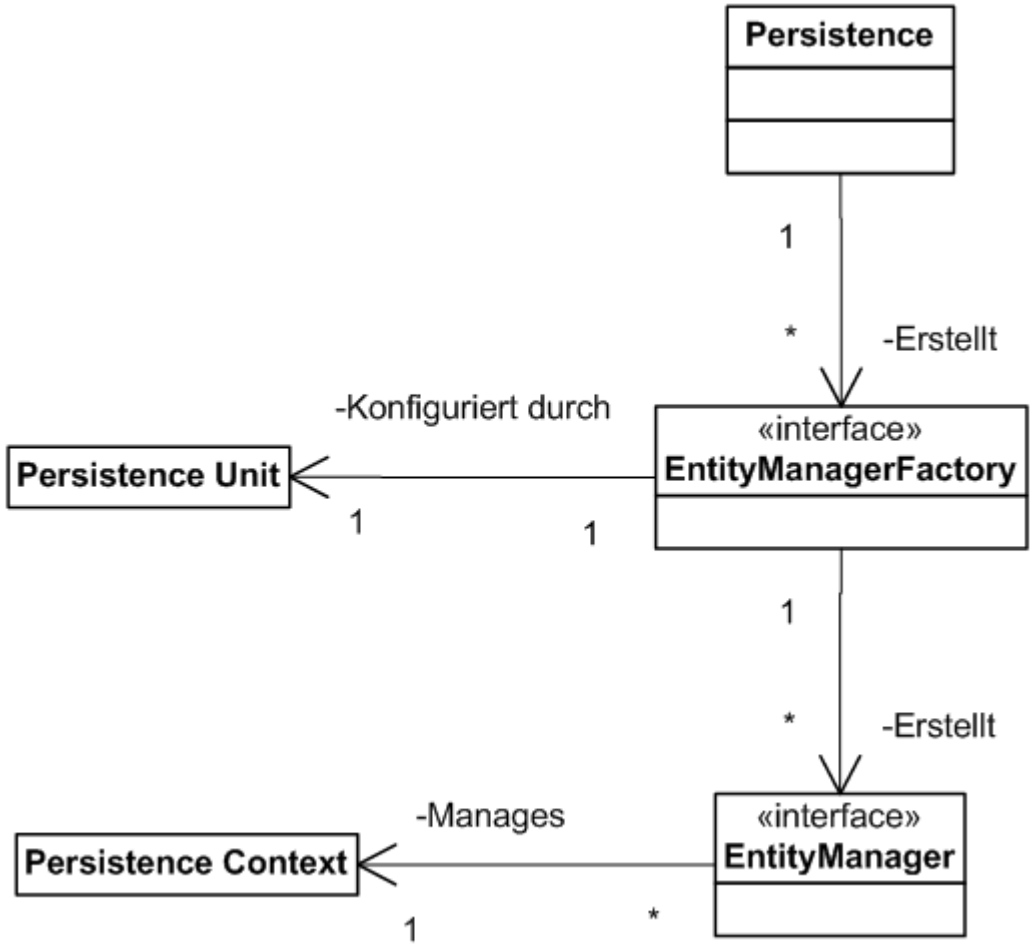
- **Detached**
  The object has a record in the database, but is disconnected, i.e. the state is no longer synchronized with the database

- **Removed**
  The object still exists, but is marked for deletion

ENTITY STATE TRANSITIONS



ENTITY MANAGER OVERVIEW



ENTITY MANAGER EXAMPLE

```
// CREATE ENTITY MANAGER
EntityManagerFactory emf = Persistence.createEntityManagerFactory("hr");
EntityManager em = emf.createEntityManager();

// PERSIST ENTITY
em.getTransaction().begin();
Employee employee = new Employee();
employee.setId(158);
em.persist(employee);
em.getTransaction().commit();

// FIND ENTITY
Employee employee = em.find(Employee.class, 158);
```

ENTITY MANAGER EXAMPLE

```
// CHANGE ENTITY
em.getTransaction().begin();
employee.setSalary(emp.getSalary() + 1000);
em.getTransaction().commit();

// DELETE ENTITY
em.getTransaction().begin();
em.remove(employee);
em.getTransaction().commit();
```

PERSISTENCE CONTEXT

The persistence context is the runtime environment of the O/R mapping and contains

- the set of all managed entities
- the used entity manager
- the current transaction
- the context type

PERSISTENCE UNIT

The persistence unit is defined by the deployment descriptor persistence.xml

```
<persistence version="3.0" ...>
  <persistence-unit name="hr" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>hr.Employee</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/hr"/>
      <property name="javax.persistence.jdbc.user" value="postgres"/>
      <property name="javax.persistence.jdbc.password" value="postgres"/>
      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

3 OBJECT-RELATIONAL MAPPING

ENTITY MAPPING

Table and column names are taken from the class and field names, but can be overridden by annotations

```
@Entity
@Table(name = "EMP")
public class Employee {
  @Id
  @Column(name = "EMP_ID")
  private int id;
  ...
}
```

PERSISTENT DATA TYPES

The following data types can be used in entities:

- Strings, primitive types, wrapper classes (e.g. Integer), BigDecimal, Date, Calendar
- Enumerations
- Arrays of byte, Byte, char, Character
- References to other entities and collections

DATA TYPE MAPPING

- Implicitly by JDBC data type conversion tables
- Explicitly by @Column annotation, e.g.

```
@Column(length = 10, nullable = false)
private String isbn;
@Column(columnDefinition = "VARCHAR(40)")
private String email;
```

- Product specific (JPA implementation, JDBC driver)

TEMPORAL TYPES

- Permitted date/time types

```
java.sql.Date/Time/Timestamp
java.util.Date/Calendar
```

- For the java.util types, the JDBC type must be specified, e.g.

```
@Temporal(TemporalType.DATE)
private Calendar dob;
```

- Support for Java 8 Date/Time API (since JPA 2.2)

```
java.time.LocalDate/LocalTime/LocalDateTime
```

ENUMERATIONS

Enumerations can be persisted as ordinal number (position) or as string (name)

```
@Enumerated(EnumType.ORDINAL)
private Color color;

@Enumerated(EnumType.STRING)
private Color color;
```

LARGE OBJECTS

Data can be stored as binary large object (blob) or character large object (clob)

```
@Lob
private byte[] picture;

@Lob
private char[] largeText;
```

TRANSIENT PROPERTIES

Fields can be excluded from persistence by modifier or annotation

```
private transient String translatedName;

@Transient
private String translatedName;
```

THE PRIMARY KEY

- Each entity class must have a field annotated with @Id

- An ID field can have the following types:
  - String, BigInteger, Date, UUID
  - Primitive Java types: byte, int, short, long, char
  - Wrapper classes: Byte, Integer, Short, Long, Character
  - Array of primitive types or wrapper classes

PRIMARY KEY GENERATION

- Primary keys can be generated in conjunction with the database

```
@Entity public class Employee {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  public Integer id;
  ...
}
```

- Generation strategies are Identity, Sequence, Table and Auto

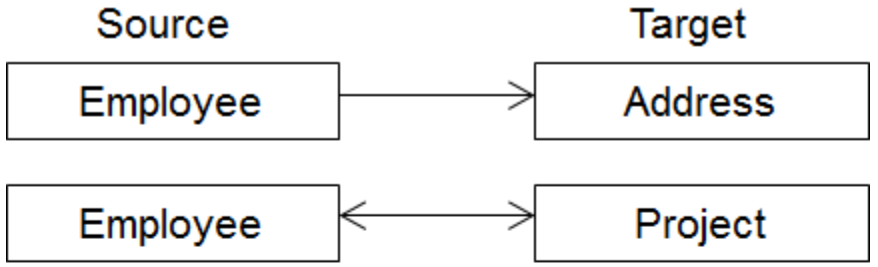
4 ENTITY RELATIONSHIPS

RELATIONSHIPS

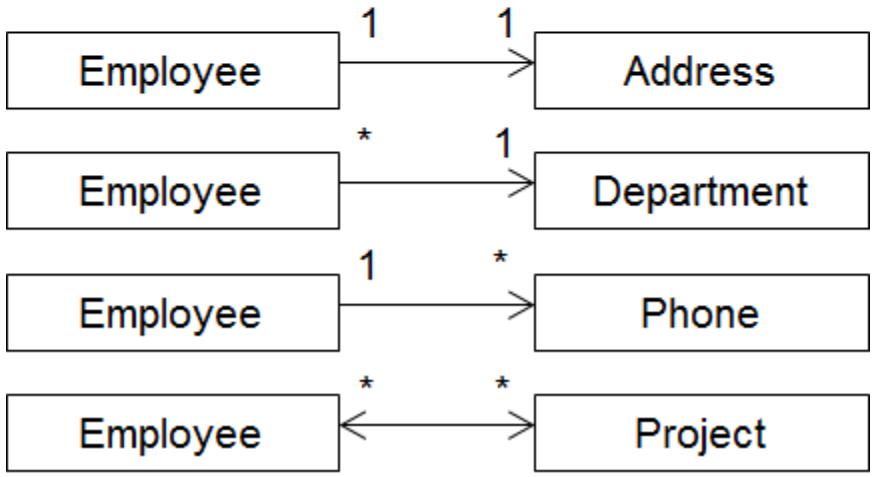
- Relationships between entities are represented by references or collections in the entity classes
- Relationships must be explicitly declared using annotations
- Additional details are often required for O/R mapping and behavior

RELATIONSHIP CHARACTERISTICS

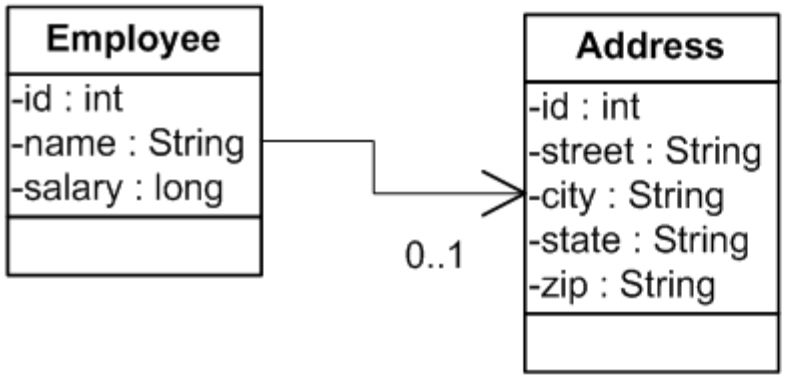
- Direction
  - unidirectional
  - bidirectional

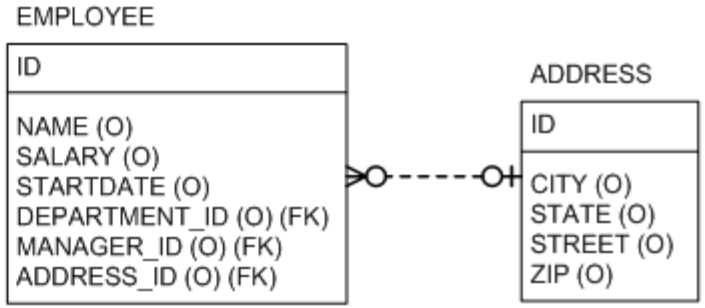


- Cardinality
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many



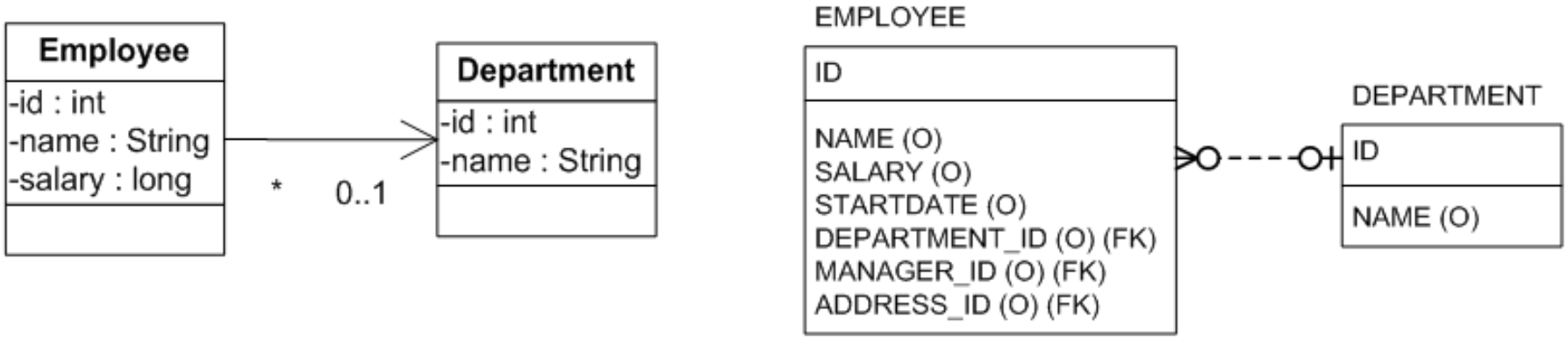
ONE-TO-ONE, UNIDIRECTIONAL





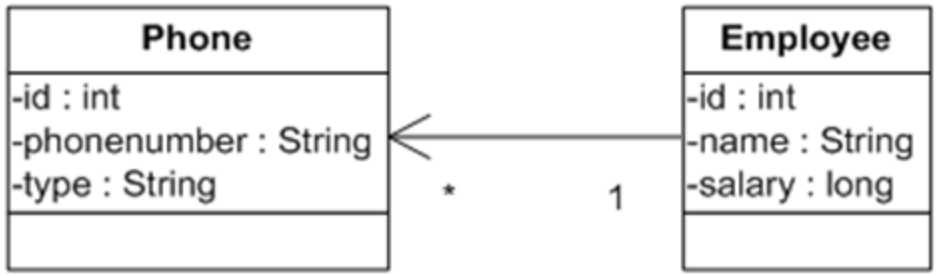

```
// Employee class
@OneToOne
private Address address;
```

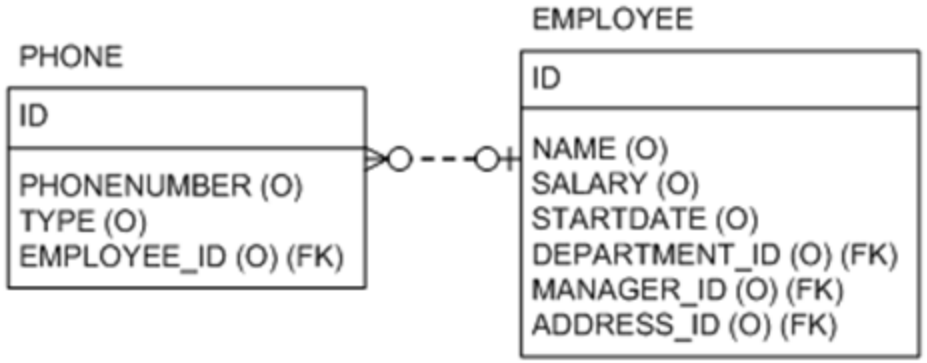
MANY-TO-ONE, UNIDIRECTIONAL




```
// Employee class
@ManyToOne
private Department department;
```

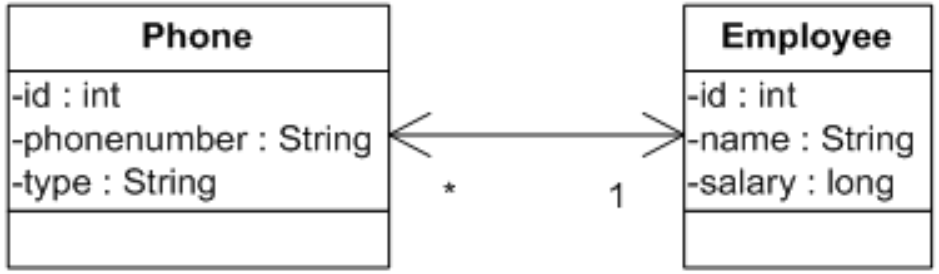
ONE-TO-MANY, UNIDIRECTIONAL

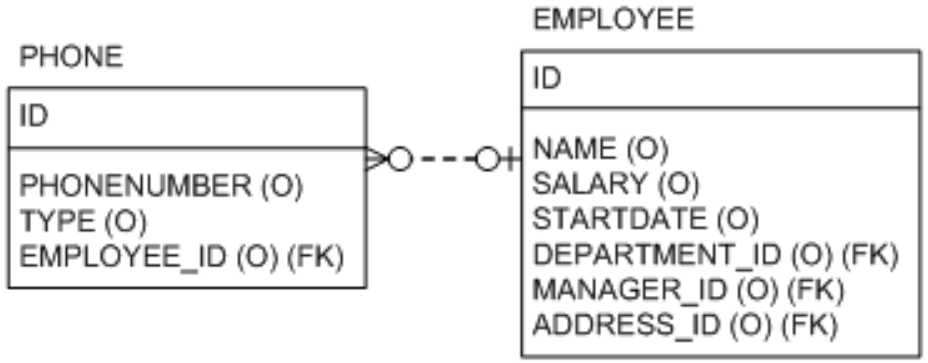





```
// Employee class
@OneToMany
@JoinColumn(name = "employee_id")
private Set<Phone> phones;
```

ONE-TO-MANY, BIDIRECTIONAL



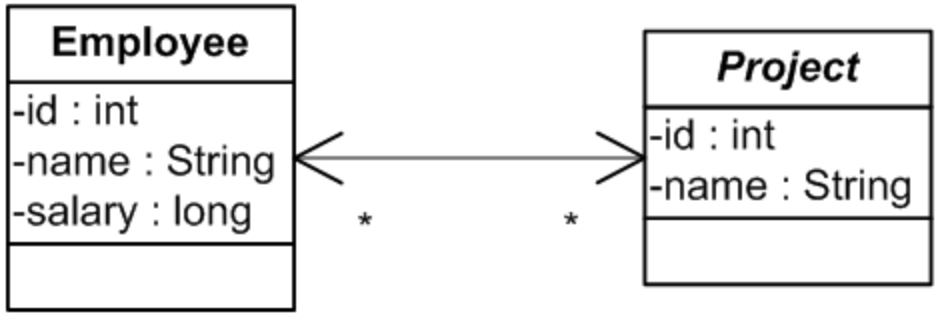


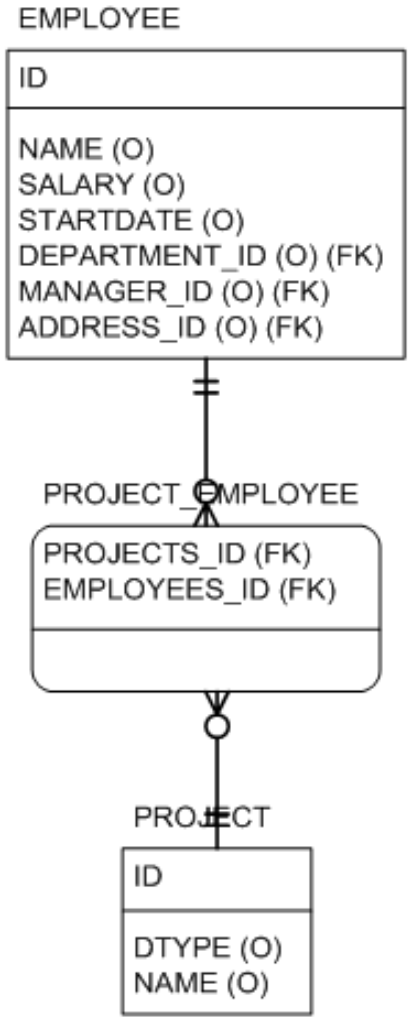

```
// Employee class (inverse side)
@OneToMany(mappedBy = "employee")
private Set<Phone> phones;

// Phone class (owning side)
@ManyToOne(optional = false)
private Employee employee;
```

OWNING AND INVERSE SIDE

- JPA distinguishes between the owning and the inverse side of a relationship:
  - The owning side is responsible for managing the relationship in the database and has the foreign key
  - The inverse side has a **mappedBy** attribute that specifies the foreign key attribute of the owning side
- In unidirectional relationships, the inverse side is missing

MANY-TO-MANY, BIDIRECTIONAL






```
// Employee class (inverse side)
@ManyToMany(mappedBy = "employees")
private Set<Project> projects;

// Project class (owning side)
@ManyToMany
private Set<Employee> employees;
```

CASCADED PERSISTENCE

- JPA supports cascaded persistence, i.e. the objects that are reachable from an entity can be included in the persistence operations

```
Employee employee = new Employee();
employee.setAddress(new Address(...));
em.persist(emp);
```

- Cascading must be declared in the relationship

```
@OneToOne(cascade = {CascadeType.PERSIST,CascadeType.REMOVE})
private Address address;
```

- Cascading types are PERSIST, MERGE, REMOVE, REFRESH, DETACH or ALL

ORPHAN REMOVAL

- Child elements in to-many relationships can be automatically deleted when they are removed from the parent entity

```
@OneToMany(mappedBy="customer",
           cascade = CascadeType.ALL, orphanRemoval = true)
private Set<Phone> phones;
```

LAZY LOADING

- Lazy loading allows to load referenced entities only when they are needed

```
@OneToMany(fetch = FetchType.LAZY)
private Set<Phone> phones;
```

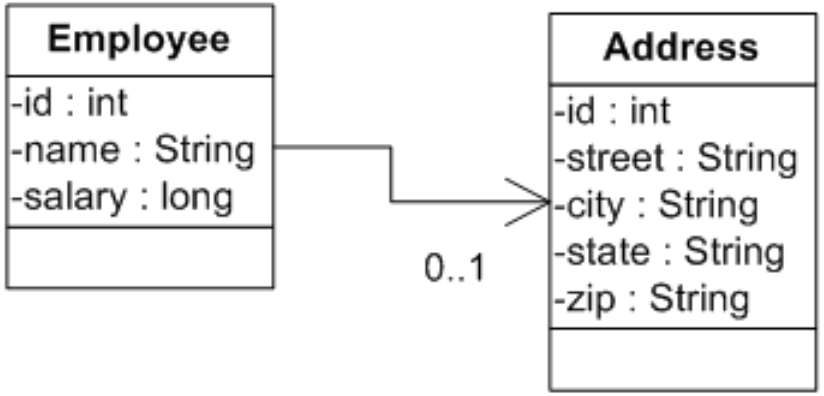
- Default fetch type is EAGER for to-one relationships and LAZY for to-many relationships

- Lazy loading does not work over transaction boundaries (e.g. in the client), so explicit queries with join fetch or entity graphs must be used

5 ADVANCED O/R MAPPING

EMBEDDED OBJECTS

- Embedded objects
  - are one way to implement composition
  - do not have their own identity
  - are in the same table as the parent object



EMPLOYEE	
PK	ID
	NAME SALARY STREET CITY STATE ZIP

EMBEDDED OBJECTS EXAMPLE

```
@Embeddable
public class Address {
  private String street;
  private String city;
  private String state;
  private String zip;
}
```

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

  @Embedded
  private Address address;
}
```

COMPOSITE PRIMARY KEY

- Composite keys are represented by their own class

```
public class EmployeeId
       implements Serializable {
  private String country;
  private int id;
  // equals and hashCode methods
}
```

- Mapping option 1

```
@IdClass(EmployeeId.class)
@Entity public class Employee {
  @Id private String country;
  @Id private int id;
  ...
}
```

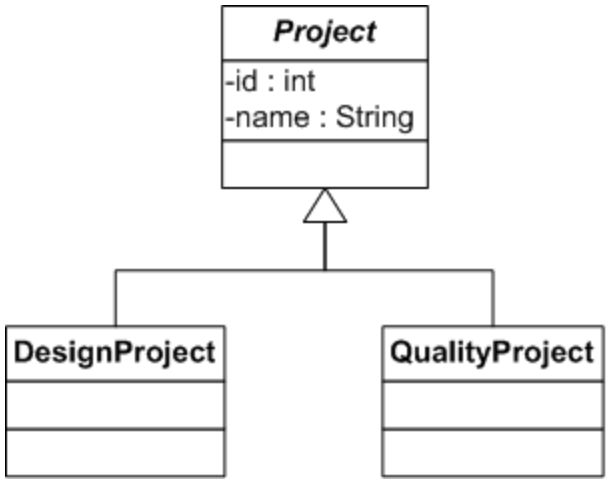
- Mapping option 2

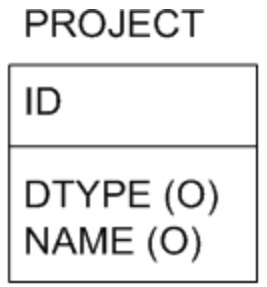
```
@Entity public class Employee {
  @EmbeddedId
  private EmployeeId id;
  ...
}
```

INHERITANCE

- Inheritance can be mapped, and base classes can be abstract
- All classes in an inheritance hierarchy inherit the primary key of the base class
- There are different mapping strategies for the database:
  - a single table for the entire inheritance hierarchy
  - a table for each non abstract class
  - a table for each class
  - mapped superclass

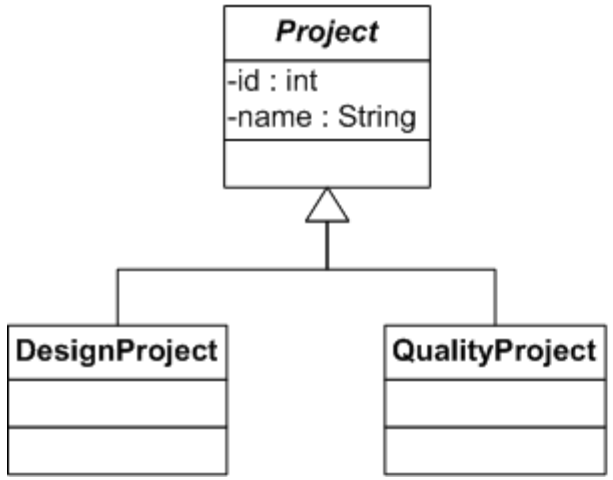
SINGLE TABLE

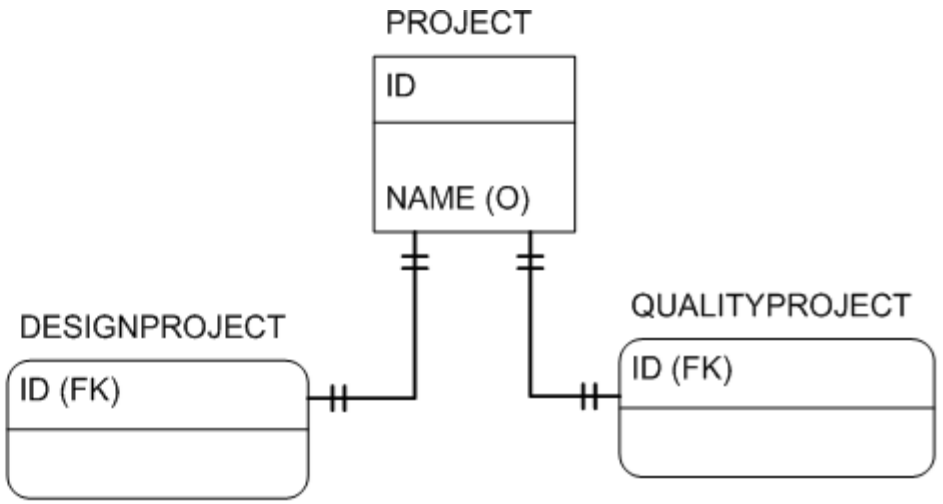





```
@Entity @Inheritance
public abstract class Project { ... }

@Entity public class DesignProject extends Project { ... }
@Entity public class QualityProject extends Project { ... }
```

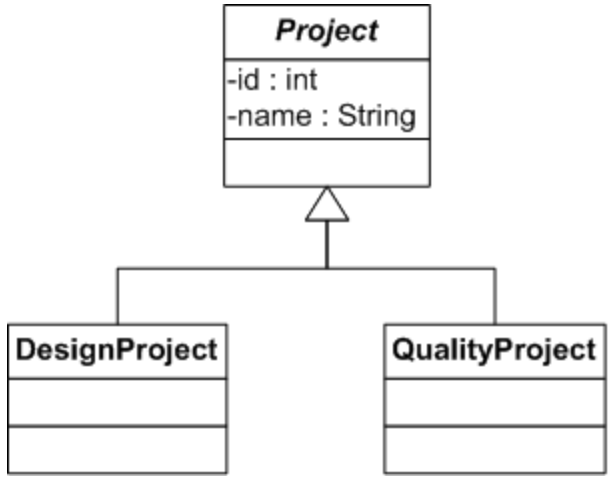
JOINED TABLE

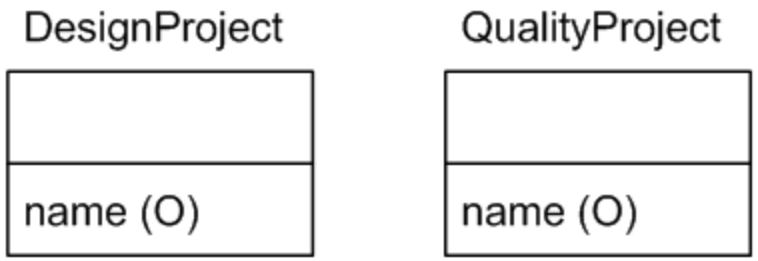





```
@Entity @Inheritance(strategy = InheritanceType.JOINED)
public abstract class Project { ... }

@Entity public class DesignProject extends Project { ... }
@Entity public class QualityProject extends Project { ... }
```

TABLE PER CLASS






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

@Entity public class DesignProject extends Project { ... }
@Entity public class QualityProject extends Project { ... }
```

MAPPED SUPERCLASS

The simplest way to map inheritance is a mapped superclass that has no representation in the database

```
@MappedSuperclass
public abstract class Base {
  @Id @GeneratedValue
  protected Integer id;
  ...
}

@Entity
public class Phone extends Base {
  ...
}
```

VERSION FIELDS

A version field is used for optimistic locking, i.e. it is checked and automatically updated by each transaction

```
@Entity
public class Employee {
  @Id
  private int id;
  @Version
  private int version;
  ...
}
```

6 USING QUERIES

QUERIES IN JPA

- Java Persistence Query Language (JPQL)
  - SQL subset
  - independent of the underlying database
  - queries based on the class model
- Criteria API since JPA 2.0
- Native SQL

USING QUERIES

```
/* DEFINE DYNAMIC QUERY */
TypedQuery<Employee> query =
    em.createQuery("SELECT e FROM Employee e", Employee.class);

/* DEFINE NAMED QUERY */
@Entity
@NamedQuery(name = "findAll", query = "SELECT e FROM Employee e")
public class Employee { ... }
TypedQuery<Employee> query = em.createNamedQuery("findAll", Employee.class);

/* EXECUTE QUERY */
List<Employee> employees = query.getResultList();
```

QUERY API

- Classes
  - Query
  - TypedQuery<T>

- Query methods
  - getResultList()
  - getSingleResult()
  - executeUpdate()
  - setParameter()
  - setFirstResult()
  - setMaxResults()

QUERY EXAMPLES

- Simple Query

```
SELECT e FROM Employee e
```

- Projections

```
SELECT e.name FROM Employee e
SELECT e.department FROM Employee e
SELECT e.name, e.salary FROM Employee e
```

QUERY EXAMPLES

- Filtering

```
SELECT e FROM Employee e
WHERE e.department.name = 'QA'
```

- Joins (implicit/explicit)

```
SELECT e.name, p.number FROM Employee e, Phone p
WHERE e = p.employee AND p.type = 'Cell'

SELECT e.name, p.number FROM Employee e JOIN FETCH e.phones p
WHERE p.type = 'Cell'
```

QUERY PARAMETERS

- Using parameters (name/positional)

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

- Passing parameters

```
// NAMED
query.setParameter("dept", "QA");
query.setParameter("salary", 40000);

// POSITIONAL
query.setParameter(1, "QA");
query.setParameter(2, 40000);
```

PATH EXPRESSIONS

- Path expressions allow navigation from an object to a referenced object

```
SELECT e.address FROM Employee e
SELECT e.address.name FROM Employee e
```

- A path expression can end in a collection

```
SELECT e.projects FROM Employee e
```

- A path expression cannot navigate beyond a collection

```
SELECT e.projects.name FROM Employee e
```

QUERY RESULTS

- Possible result types are
  - primitive types and strings
  - entity types
  - array of objects
  - custom types (through constructor expressions)

- If the result is an entity, it will be in the managed state

MULTIPLE RESULTS

If a query contains a projection on multiple values, a list of object arrays is returned

```
Query query = em.createQuery(
    "SELECT e.name, e.department.name " +
    "FROM Project p JOIN p.employees e WHERE p.name = 'ZLD'");

List<Object[]> results = query.getResultList();
results.forEach(values -> System.out.println(values[0] + ", " + values[1]));
```

CONSTRUCTOR EXPRESSIONS

Constructor expressions allow returning typed results for projections on multiple values

```
public record EmployeeDTO(String name, String deptName) {}

TypedQuery<EmployeeDTO> query = em.createQuery(
    "SELECT NEW hr.dto.EmployeeDTO(e.name, e.department.name) " +
    "FROM Project p JOIN p.employees e WHERE p.name = 'ZLD'");

List<EmployeeDTO> employees = query.getResultList();
employees.forEach(e -> System.out.println(e.name() + ", " + e.deptName()));
```

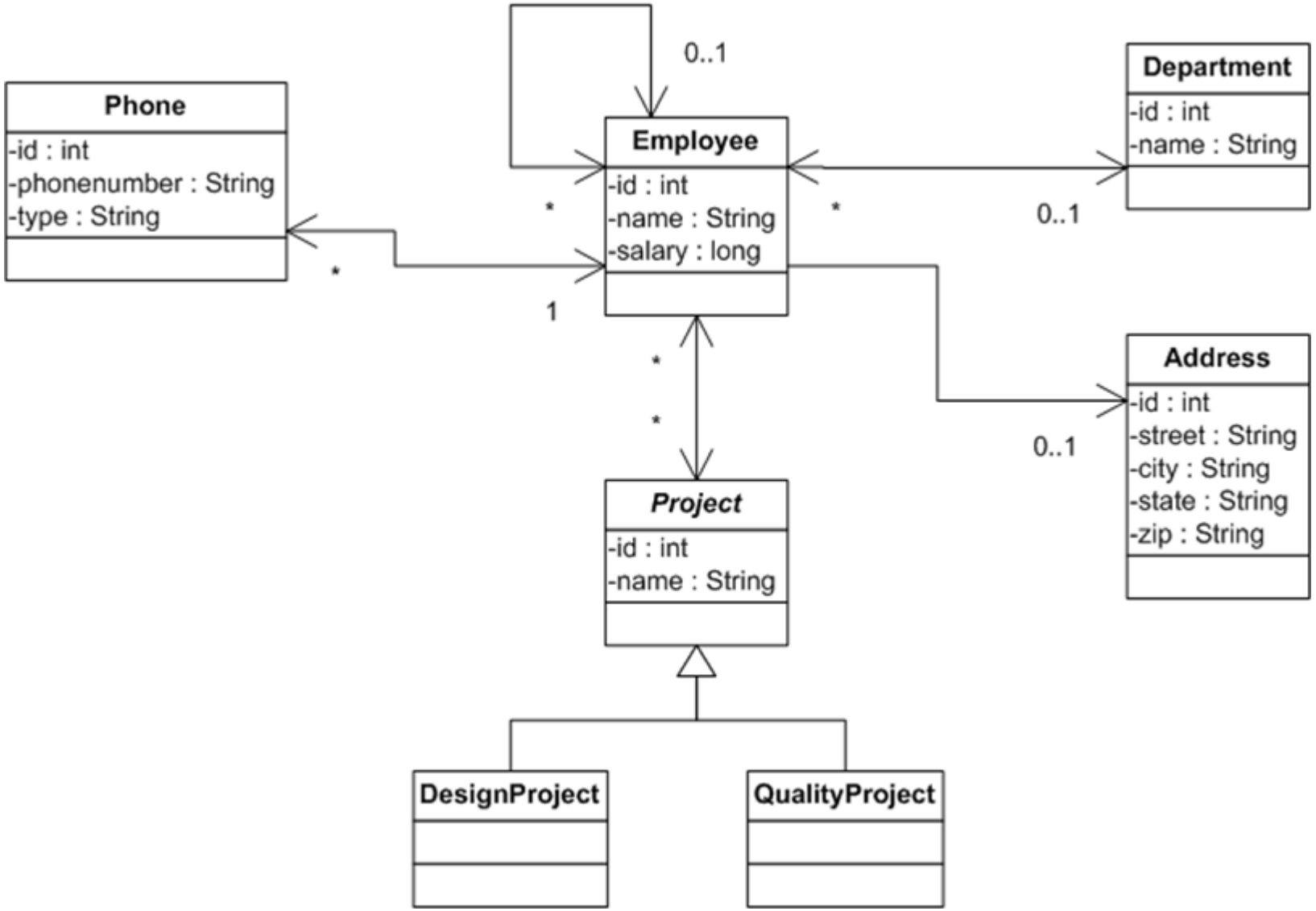
PAGING

Paging can be used to limit the result size

```
TypedQuery<Employee> query = ...
query.setFirstResult(1);
query.setMaxResults(10);

List<EmployeeDTO> employees = query.getResultList();
employees.forEach(e -> ...);
```

7 QUERY LANGUAGE

CLASS MODEL


Phone
-id : int
-phonenumber : String
-type : String
0..1
Employee
-id : int
-name : String
-salary : long
Department
-id : int
-name : String
Address
-id : int
-street : String
-city : String
-state : String
-zip : String
Project
-id : int
-name : String
DesignProject
QualityProject
*
*
1
*
*
*
*
0..1
0..1

SELECT

- A SELECT query has the following structure

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
```

- Example

```
SELECT e FROM Employee e
WHERE e.name = 'John Doe'
ORDER BY e.salary
```

JOINS

- Inner joins

```
SELECT p FROM Employee e JOIN e.phones p
```

- Outer joins

```
SELECT e, d FROM Employee e LEFT JOIN e.department d
```

- Fetch joins

```
SELECT e FROM Employee e JOIN FETCH e.address
```

WHERE CONDITIONS

- Literals
- Parameters (named/positional)
- Operators
  - Navigation (.)
  - Unary (+/-)
  - Arithmetic (+, -, *, /)
  - Comparison (=, >, >=, <, <=, NOT, BETWEEN, LIKE, IN, IS NULL, IS EMPTY, MEMBER OF)
  - Logical (AND, OR, NOT)

BETWEEN

The BETWEEN operator checks if a value is in a certain range (including limits)

```
SELECT e FROM Employee e
WHERE e.salary BETWEEN 40000 AND 45000
```

EMPTY

The EMPTY operator checks if a collection is empty (or not)

```
SELECT e FROM Employee e
WHERE e.phones IS NOT EMPTY
```

MEMBER OF

The MEMBER OF operator checks if a value or entity is member of a JPA collection

```
SELECT e
FROM Employee e
WHERE :project MEMBER OF e.projects
```

IN

The IN operator checks if a value or entity is contained in a specified set

```
SELECT e FROM Employee e
WHERE e.address.state IN ('NY','CA')
```

EXISTS

The EXISTS operator checks if a subquery returns any results

```
SELECT e FROM Employee e
WHERE NOT EXISTS (SELECT p FROM e.phones p WHERE p.type = 'Cell')
```

ALL, ANY

The ALL and ANY operators check if a condition is satisfied for all or some results of the subquery

```
SELECT e FROM Employee e
WHERE e.salary >= ALL (SELECT c.salary FROM e.department.employees c)
```

FUNCTIONS

Values can be processed in the select clause using the following functions

- Strings
  CONCAT, LENGTH, LOCATE, LOWER, SUBSTRING, UPPER, TRIM
- Numbers
  ABS, MOD, SQRT
- Date/Time
  CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP
- Collections
  SIZE, AVG, COUNT, MAX, MIN, SUM

ORDER BY

The ORDER BY clause can be used to sort the results of a query by a value contained in the select clause

```
SELECT e FROM Employee e
ORDER BY DESC e.name
```

```
SELECT e FROM Employee e
ORDER BY e.name, e.salary DESC
```

GROUP BY

The GROUP BY clause defines a grouping for the aggregation or results

```
SELECT d.name, COUNT(e)
FROM Department d JOIN d.employees e
GROUP by d
```

HAVING

The HAVING clause defines a filter that is used for the grouping of results

```
SELECT e.name
FROM Employee e JOIN e.projects p
GROUP BY e HAVING COUNT(p) > 1
```

UPDATE

- An UPDATE query has the following structure

```
UPDATE <entity_name> [[AS] <identification_variable>]
SET <update_statement> {, <update_statement>}*
[WHERE <conditional_expression>]
```

- Example

```
UPDATE Employee e
SET e.salary = 60000
WHERE e.salary = 55000
```

DELETE

- A DELETE query hat the following structure

```
DELETE FROM <entity_name>
[WHERE <conditional_expression>]
```

- Example

```
DELETE FROM Employee e
WHERE e.department IS NULL
```