Taller de Sistemas de Información 2

Persistencia

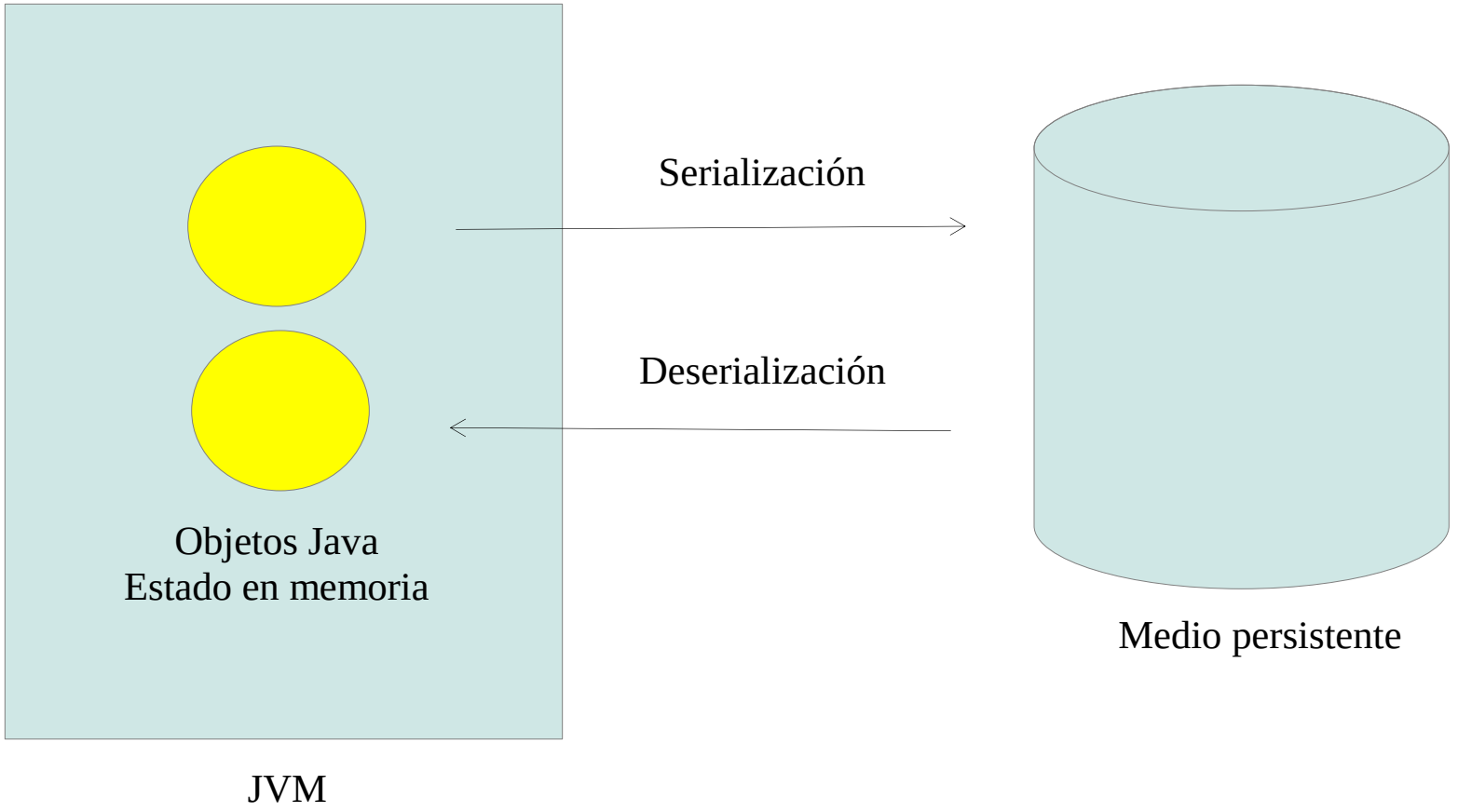
28 de Agosto de 2014

Instituto de Computación
Facultad de Ingeniería
Universidad de la República de Uruguay

Persistencia

- Es una característica de los procesos de un computador, en la cual el estado de los mismos, sobrevive al proceso que lo creo
- Sin esta característica, el estado solo existiría en memoria, perdiéndose cuando la misma es vaciada
- Esto se logra en la practica, almacenando dicho estado en un medio no volátil, como un disco duro o una memoria flash

Persistencia en Java



Persistencia en Java

- Soluciones propietarias
  - Serialización/Deserialización binaria / XML
- Java Database Connectivity
- EJB Entity Beans (EJB 2.1)
- ORM Mapper
  - Hibernate, Toplink
- Java Data Objects
- Java Persistence API

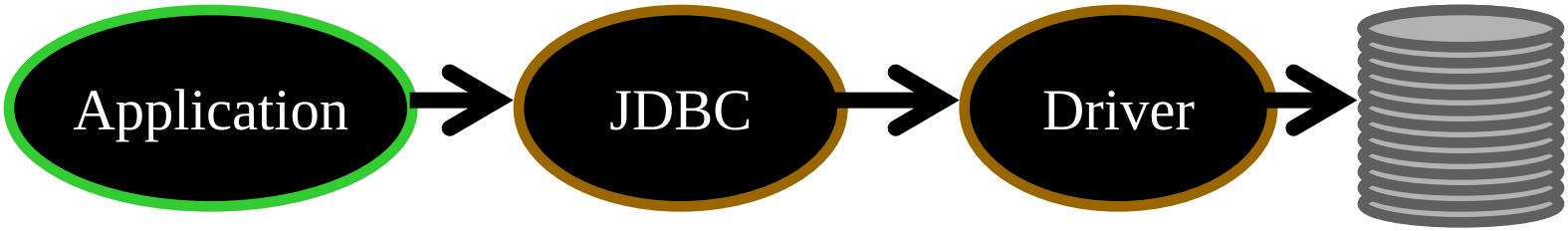
JDBC

- **Java Data Base Connectivity** API

- Diseñado para simplificar tareas diarias con la base de datos
- Permite codificar consultas de acceso y modificación a la base de datos en SQL
- Provee una arquitectura basada en drivers e interfaces estándar

Arquitectura de JDBC

- El codigo Java invoca una biblioteca JDBC
- JDBC carga a la JVM un “driver”
- El “driver” se encarga de hablar con un DBMS particular



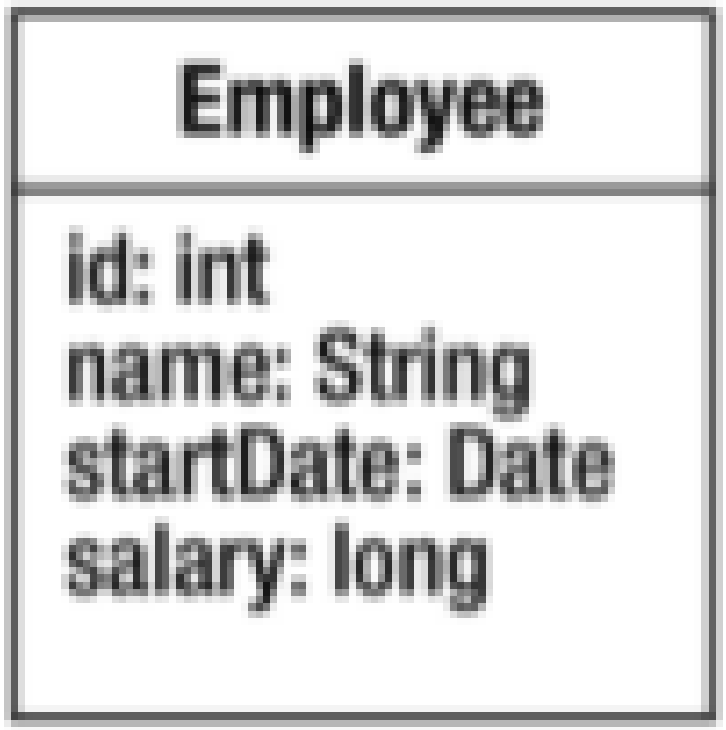
Object – Relational mapping

- La técnica que permite llenar el vacío entre el modelo relacional y el modelo orientado a objetos, de denomina **object relational mapping**
  - También conocido como O-R mapping u ORM
- Introducimos un mediador, encargado de mapear automáticamente los conceptos de un modelo en el otro

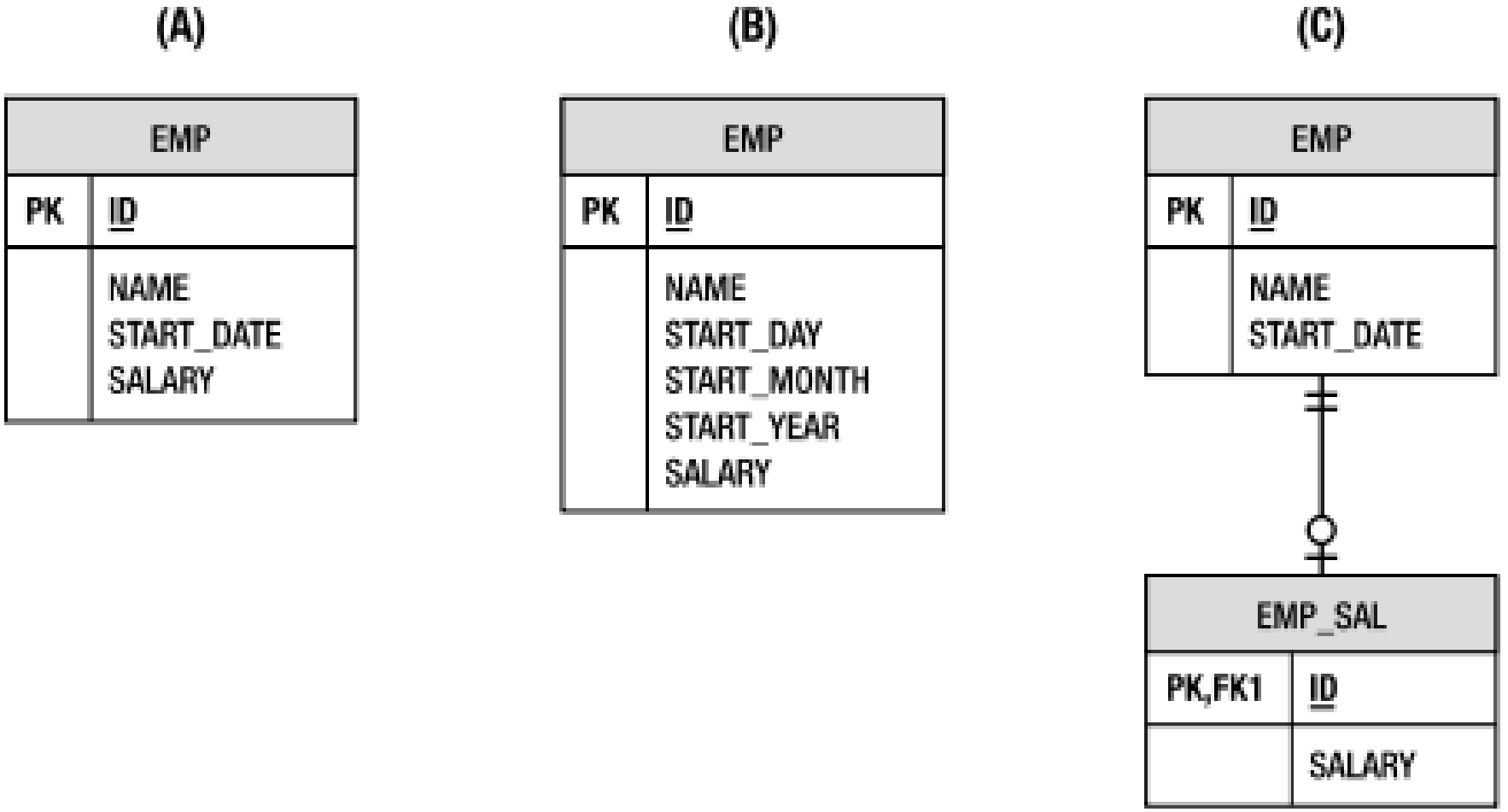
Impedance Mismatch

- La diferencia entre el modelo relacional y el modelo orientado a objetos, se denomina Impedance Mismatch
- El problema de mapear uno en otro, no surge debido a las similaridades, sino a las diferencias entre ambos modelos
- Veamos algunos ejemplos...

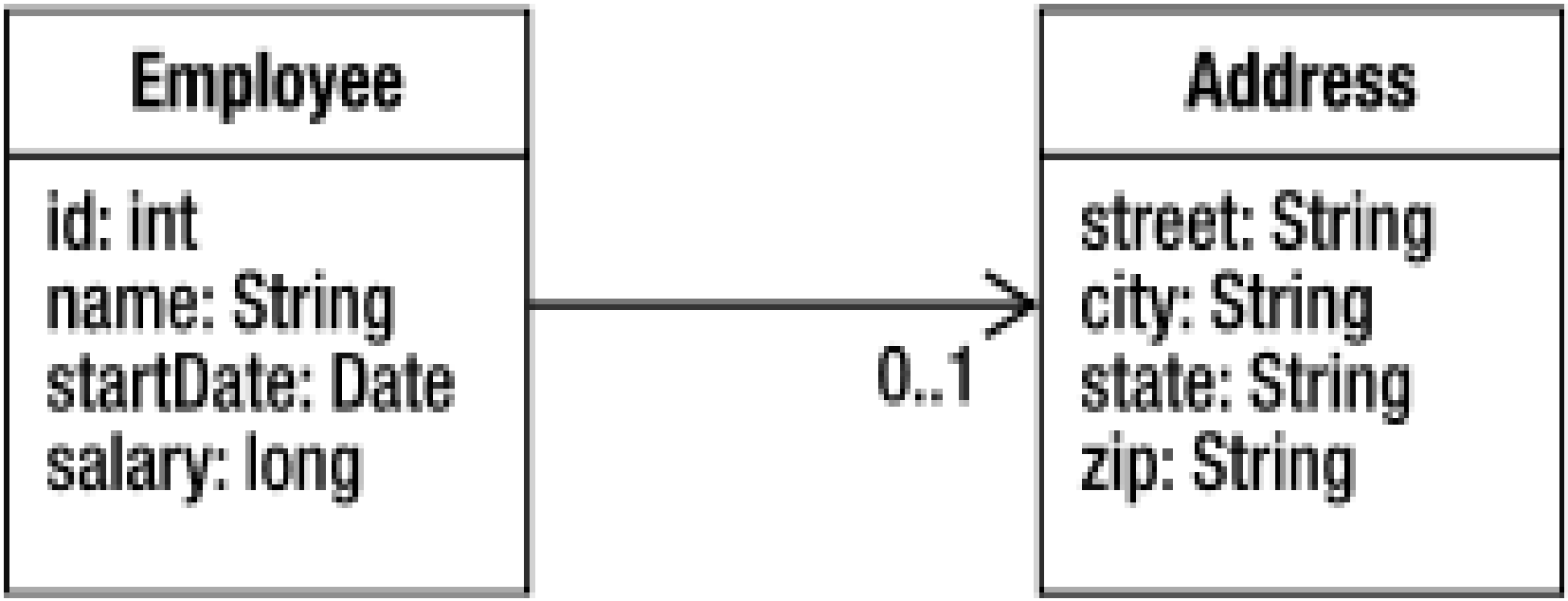
Impedance Mismatch



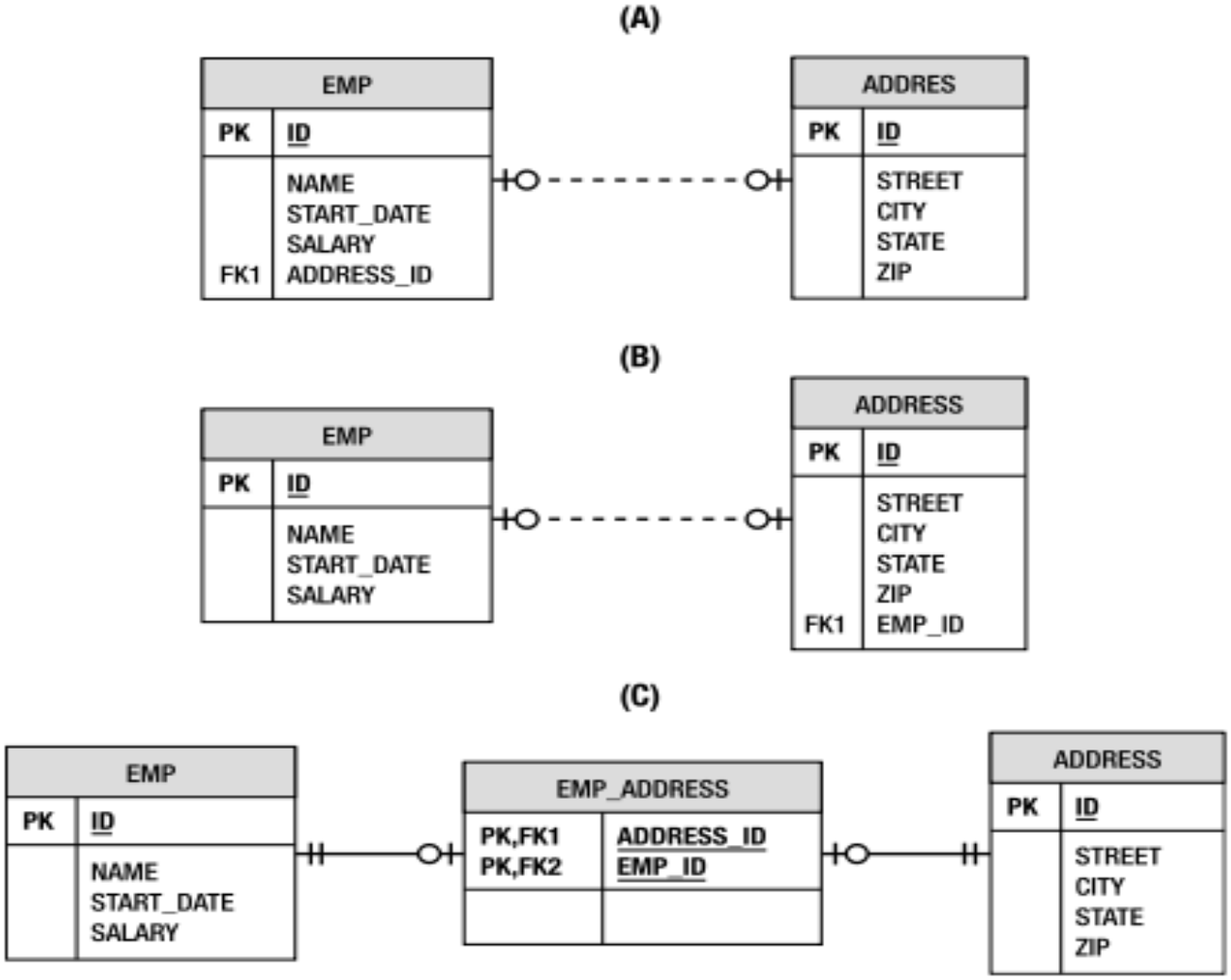
Impedance Mismatch



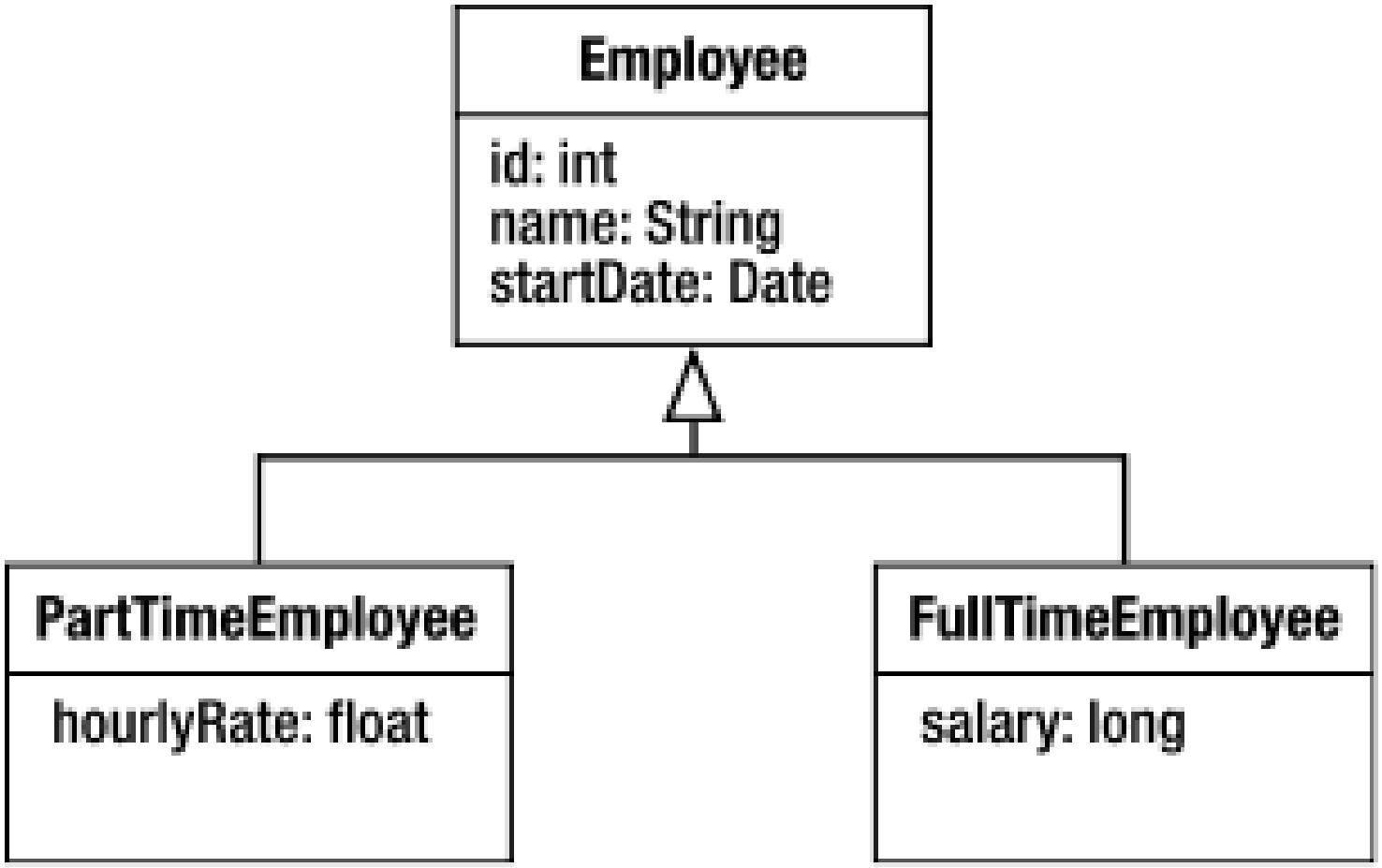
Impedance Mismatch



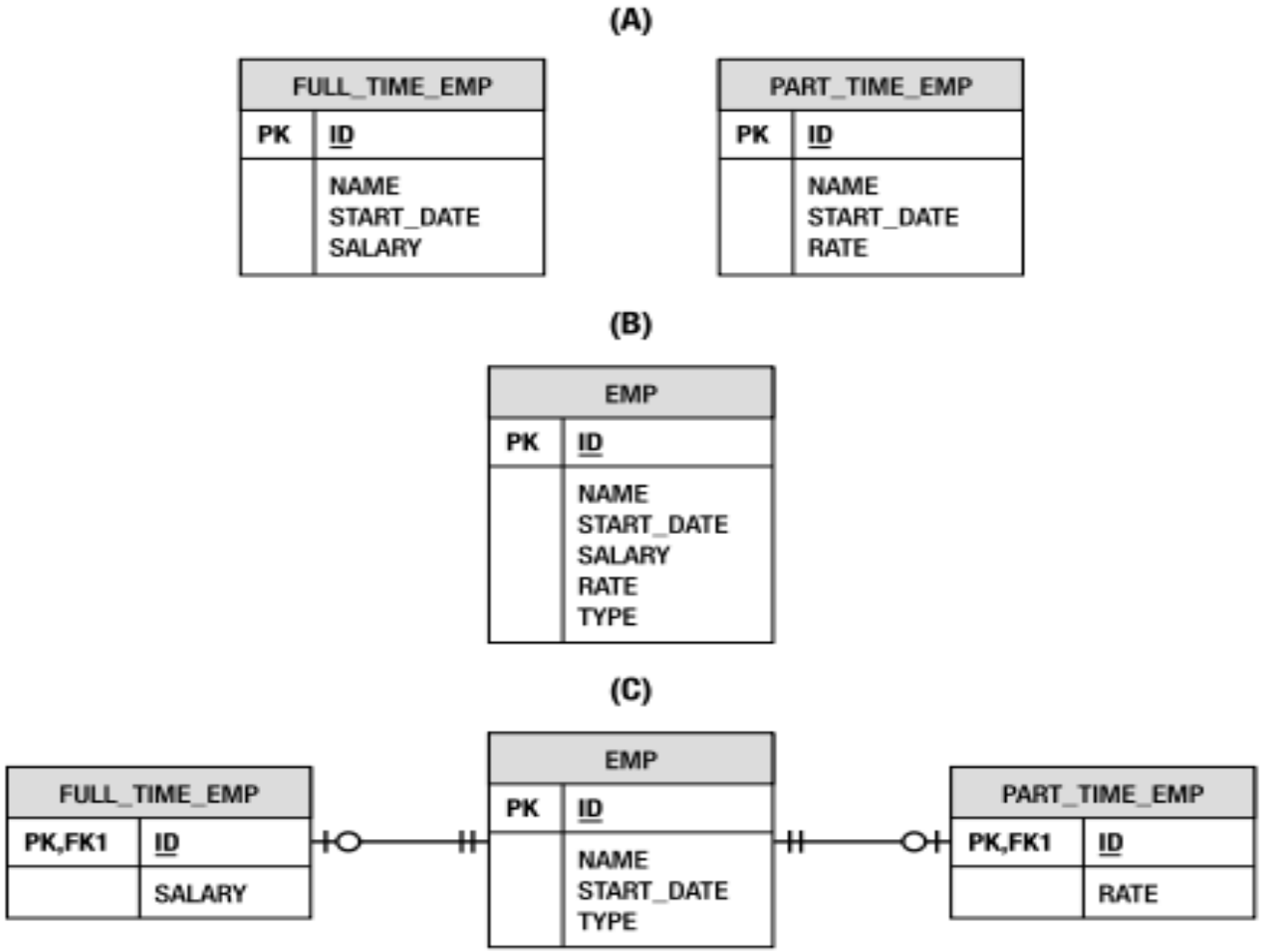
Impedance Mismatch



Impedance Mismatch



Impedance Mismatch



Java Persistence API

- Es un framework liviano para persistencia basada en POJOs
- Ofrece una buena solución para integrar soporte de persistencia en aplicaciones de todo tipo
- Surge originalmente de la mano de EJB 3.0, adoptando la experiencia de años de uso de Hibernate y Toplink, seguidos luego por JDO

Entidades

- Es un “sustantivo” en nuestro modelo de negocio
- Es una agrupación de estado, asociado como una sola unidad
- Una entidad puede participar en cualquier numero de relaciones con otras entidades
- En el mundo OO, se le puede agregar comportamiento
  - Es lo que generalmente denominamos Objetos

Entidades

- En JPA cualquier objeto de aplicación puede ser una entidad
- Existen una serie de características que determinan que dicho objeto pueda ser denominado “entidad”
  - Persistibles
  - Identidad
  - Transaccionalidad
  - Granularidad apreciable

Metadatos

- Además del estado persistente, las entidades tienen asociado un conjunto de metadatos
- Existen dos formas de definir metadatos, en forma de Anotaciones Java y en forma de XML
- Los metadatos en JPA siguen el estilo de **configuración por excepción**
  - Se utilizan valores por defecto que aplican a la mayoría de las aplicaciones
  - Solo debemos configurar, cuando se aparta del valor por defecto

Clase Employee

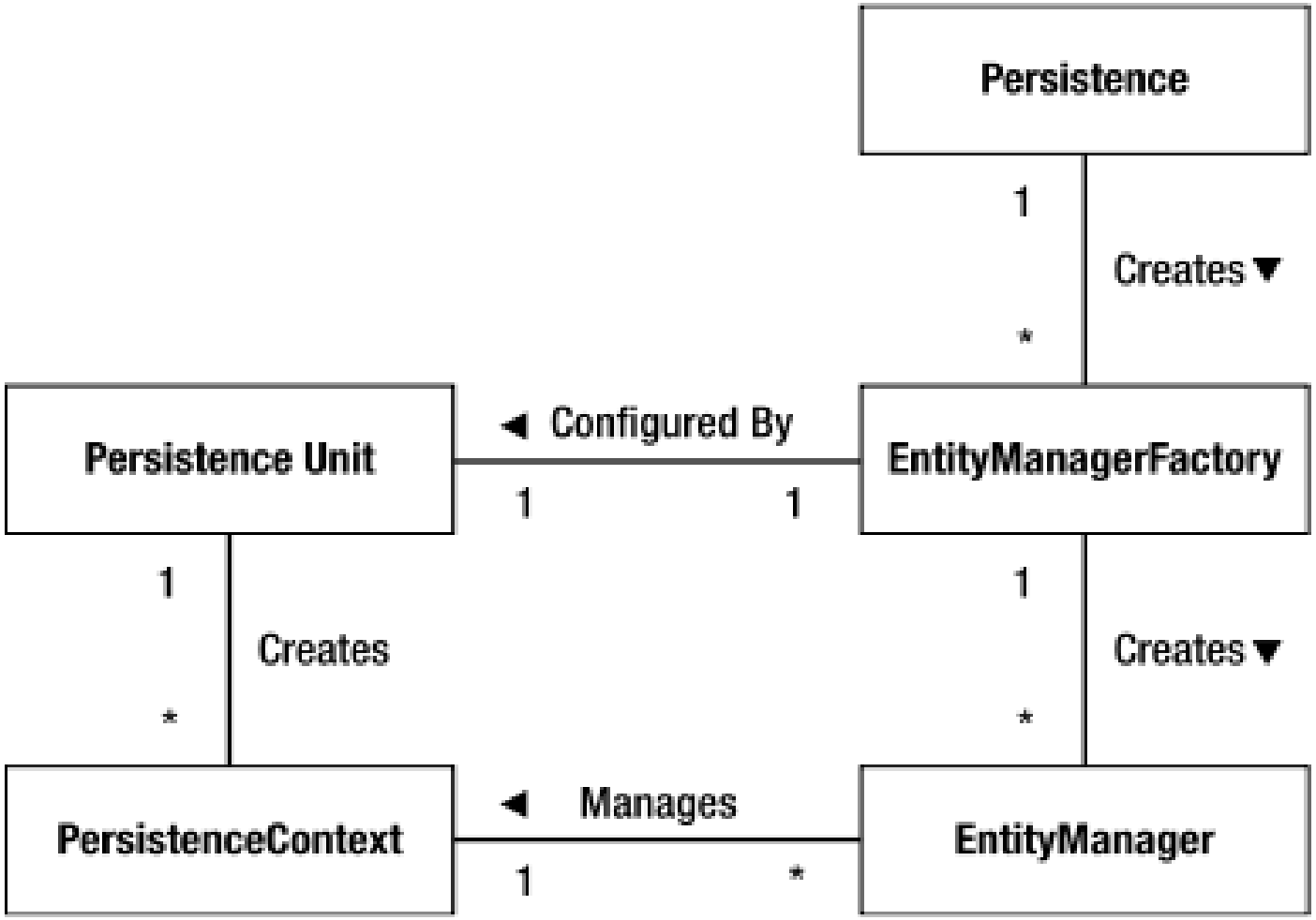
```
public class Employee {
    private int id;
    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

Entidad Employee

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

    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

Conceptos de JPA



Conceptos de JPA

- El API de persistencia se encuentra implementada en una clase denominada EntityManager, representada por la interfaz EntityManager
- Los EntityManager son producidos por instancias de EntityManagerFactory
- Una Persistence Unit define las entidades utilizadas o administradas por un EntityManager

Obteniendo un EntityManager

- Existen diversas formas de obtener un EntityManager, según si estamos en Java SE o Java EE
- En Java SE, podemos usar la clase de bootstrap **Persistence**

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("EmployeeService");

EntityManager em = emf.createEntityManager();
```

Persistiendo una entidad

- Luego de que tenemos el EntityManger instanciado, podemos empezar a usar las funciones que este provee

```
public Employee createEmployee(int id, String name, long salary) {
    Employee emp = new Employee(id);
    emp.setName(name);
    emp.setSalary(salary);
    em.persist(emp);
    return emp;
}
```

Buscando y removiendo...

```
public Employee findEmployee(int id) {
    return em.find(Employee.class, id);
}


public void removeEmployee(int id) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        em.remove(emp);
    }
}
```

Actualizando una entidad

- Una vez que la entidad esta asociada al EntityManager, podemos realizar actualizaciones automáticas..

```
public Employee raiseEmployeeSalary(int id, long raise) {
    Employee emp = em.find(Employee.class, id);

    if (emp != null) {
        emp.setSalary(emp.getSalary() + raise);
    }

    return emp;
}
```

Transacciones

- Según el contexto en el que ejecuta una aplicación, la forma de iniciar/terminar transacciones puede variar
- En Java SE, debemos hacer esto explícitamente

```
em.getTransaction().begin();

createEmployee(158, "John Doe", 45000);

em.getTransaction().commit();
```

Consultas

- En algún momento en nuestra aplicación, deberemos utilizar consultas sobre la base de datos para recuperar información
- En JPA, utilizamos consultas orientadas a objetos, aplicadas sobre las entidades, sus propiedades y sus relaciones
- Este lenguaje se denomina Java Persistence Query Language (JPQL)
- Las consultas pueden ser de tipo Query, obteniéndose a través de la interfaz EntityManager

Consultas

- Por ejemplo, un método para recuperar los empleados de la aplicación

```
public List<Employee> findAllEmployees() {

    TypedQuery<Employee> query = em.createQuery(
          "SELECT e FROM Employee e", Employee.class);

    return query.getResultList();
}
```

```
import javax.persistence.*;
import java.util.List;

public class EmployeeService {
    protected EntityManager em;
    public EmployeeService(EntityManager em) {
        this.em = em;
    }
    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }
    public void removeEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null)
            em.remove(emp);
    }
    public Employee raiseEmployeeSalary(int id, long raise) {
        Employee emp = em.find(Employee.class, id);
        if (emp != null)
            emp.setSalary(emp.getSalary() + raise);
        return emp;
    }
    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }
    public List<Employee> findAllEmployees() {
        TypedQuery<Employee> query = em.createQuery(
                "SELECT e FROM Employee e", Employee.class);
        return query.getResultList();
    }
}
```

Empaquetado

- Con los elementos anteriores, tenemos las piezas basicas de una aplicación JPA
- Debemos combinar dichas piezas para lograr una aplicación que funcione primero en Java SE
- Para esto, lo primero a definir es la Persistence Unit

Persistence Unit

- Se define un archivo XML denominado persistence.xml
- Cada Persistence Unit tiene un nombre único, el cual es usado por la aplicación para referirse a la misma
- Dentro del archivo persistence.xml pueden definirse todas las unidades que se requieran, siendo cada una de estas independiente

persistence.xml

```
<persistence>
    <persistence-unit name="EmployeeService"
                      transaction-type="RESOURCE_LOCAL">
        <class>examples.model.Employee</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
                      value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="javax.persistence.jdbc.user" value="APP"/>
            <property name="javax.persistence.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

Persistence Archive

- Los artefactos de persistencia son empaquetados en lo que denominamos un Persistence Archive
- Es un archivo JAR tradicional que:
  - Contiene el archivo persistence.xml dentro de la carpeta META-INF
  - Contiene los .class de las entidades que se persistirán

Ejecutando la aplicación

- Solo necesitamos colocar en el classpath los siguientes elementos:
  - El JAR de la aplicación con el Persistence Archive
  - Los JARs de JPA
  - Los JARs de los proveedores de persistencia
  - El driver de conexión a la base de datos

ORM

- Uno de los aspectos mas importantes de cualquier solución de ORM, es la configuración que permite mapear el estado de las entidades en la base de datos
- Esta información de configuración se presenta en forma de anotaciones, las cuales son colocadas dentro de las entidades

Mapeando a una tabla

- Cuando queremos mapear una entidad a una tabla especifica, entonces debemos utilizar la anotación @Table
- Esto debe hacerse, cuando el nombre de la tabla no coincide con el de la entidad

```
@Entity
@Table(name="EMP")
public class Employee { ... }
```

Mapeando a una tabla

- También podemos definir una esquema o un catalogo asociado a una tabla

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }

@Entity
@Table(name="EMP", catalog="HR")
public class Employee { ... }
```

Mapeando tipos simples

- Primitive Java types: byte, int, short, long, boolean, char, float, double
- Wrapper classes of primitive Java types: Byte, Integer, Short, Long, Boolean, Character, Float, Double
- Byte and character array types: byte[], Byte[], char[], Character[]
- Large numeric types: java.math.BigInteger, java.math.BigDecimal

Mapeando tipos simples

- Strings: java.lang.String
- Java temporal types: java.util.Date, java.util.Calendar
- JDBC temporal types: java.sql.Date, java.sql.Time, java.sql.Timestamp
- Enumerated types: Cualquier enumerado definido por el sistema o el usuario
- Serializable objects: Cualquier tipo de datos del sistema o del usuario, serializable

Mapeando columnas

- Usamos la anotación @Column sobre las propiedades o campos

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    ...
}
```

Fetching de campos

- Podemos definir el momento en el que se cargan los campos de una entidad

```
@Entity
public class Employee {
    // ...
    @Basic(fetch=FetchType.LAZY)
    @Column(name="COMM")
    private String comments;
    // ...
}
```

Large Objects

```
@Entity
public class Employee {
      @Id
      private int id;
      @Basic(fetch=FetchType.LAZY)
      @Lob @Column(name="PIC")
      private byte[] picture;
      // ...
}
```

Tipos enumerados (ordinales)

```
public enum EmployeeType {
      FULL_TIME_EMPLOYEE,
      PART_TIME_EMPLOYEE,
      CONTRACT_EMPLOYEE
}

@Entity
public class Employee {
      @Id private int id;
      private EmployeeType type;
      // ...
}
```

Tipos enumerados (strings)

```
@Entity
public class Employee {
    @Id
    private int id;
    @Enumerated(EnumType.STRING)
    private EmployeeType type;
    // ...
}
```

Temporal types

- Se soportan java.sql.Date, java.sql.Time, java.sql.Timestamp y los tipos java.util.Date y java.util.Calendar

```
@Entity
public class Employee {
    @Id
    private int id;
    @Temporal(TemporalType.DATE)
    private Calendar dob;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}
```

Transient state

- Se utiliza la anotación @Transient para indicar que un atributo no forma parte del estado persistente de un objeto

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    transient private String translatedName;
    // ...
    public String toString() {
        if (translatedName == null) {
            translatedName =
                ResourceBundle.getBundle("EmpResources").getString("Employee");
        }
        return translatedName + ": " + id + " " + name;
    }
}
```

Primary Key

- Toda entidad persistente debe tener asociado un campo que declare la clave primaria en la tabla a la cual se mapea dicha entidad
- Las claves primarias son insertables, pero no actualizables ni eliminables
- Salvo por la característica especial de ser clave primaria, un campo de estas características, puede ser cualquier tipo simple de los mapeados por JPA

Primary Key Types

- Tipos Java primitivos: byte, int, short, long, char
- Wrapper classes de tipos Java primitivos: Byte, Integer, Short, Long, Character
- String: java.lang.String
- Large numeric type: java.math.BigInteger
- Temporal types: java.util.Date, java.sql.Date
- Se permiten tipos flotantes y BigDecimal, pero no se recomienda por la naturaleza no confiable del redondeo

Generación de identificadores

- A veces las aplicaciones deciden desentenderse de la generación automática de los identificadores
- Esto se logra a través de la anotación @GeneratedValue, usada en combinación con la anotación @Id
- Dependiendo de la forma en que se genera, no podemos confiar en que el valor este presente hasta que la transacción sea completada

Generación automática

- Si nos interesa la forma en que se generan los identificadores, sino solamente que se genere uno automáticamente, entonces podemos usar GenerationType.AUTO
- Puede requerir permisos de DBA, para poder utilizar el mecanismo apropiado (por ej. Crear una Tabla)

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    // ...
}
```

Generación usando tablas

```
@TableGenerator(name="Address_Gen",
     table="ID_GEN",
     pkColumnName="GEN_NAME",
     valueColumnName="GEN_VAL",
     pkColumnValue="Addr_Gen",
     initialValue=10000,
     allocationSize=100)
@Id @GeneratedValue(generator="Address_Gen")
private int id;
```

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0
Addr_Gen	10000

Generación usando secuencias

```
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id @GeneratedValue(generator="Emp_Gen")
private int getId;


CREATE SEQUENCE Emp_Seq
     MINVALUE 1
     START WITH 1
     INCREMENT BY 50
```

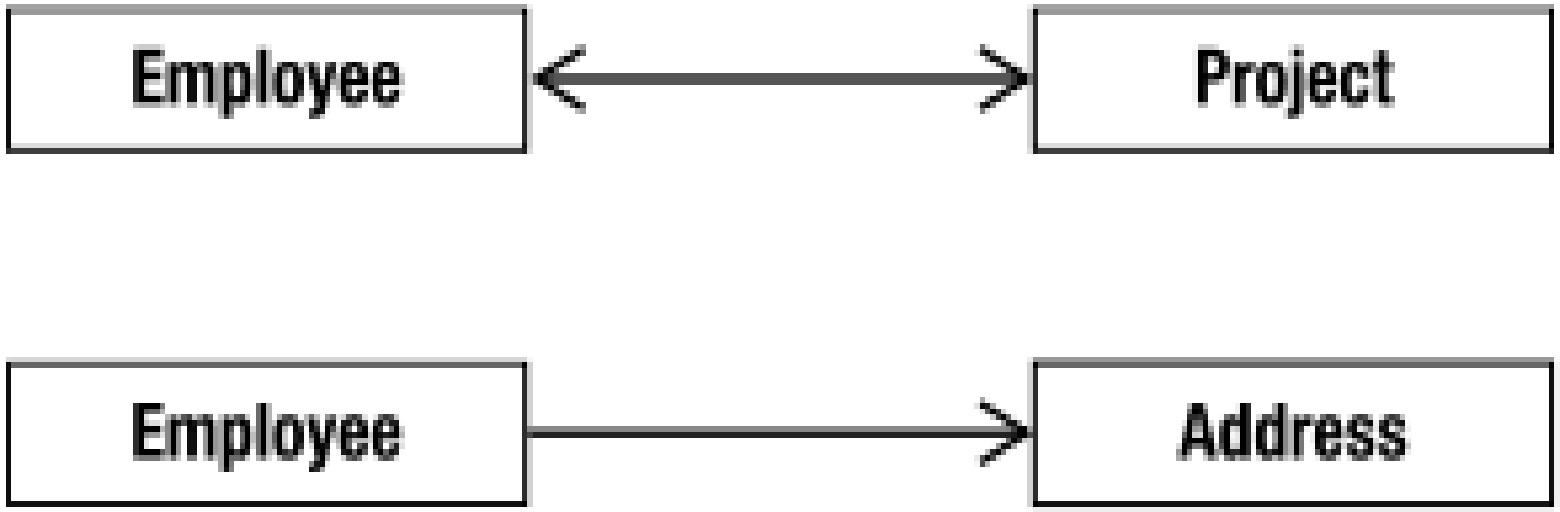
Generación usando identidades

- Se utiliza el valor GenerationType.IDENTITY
- Requiere que a nivel de esquema se soporten tipos identidad
- El identificador no esta disponible hasta luego de que se confirme la transacción

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

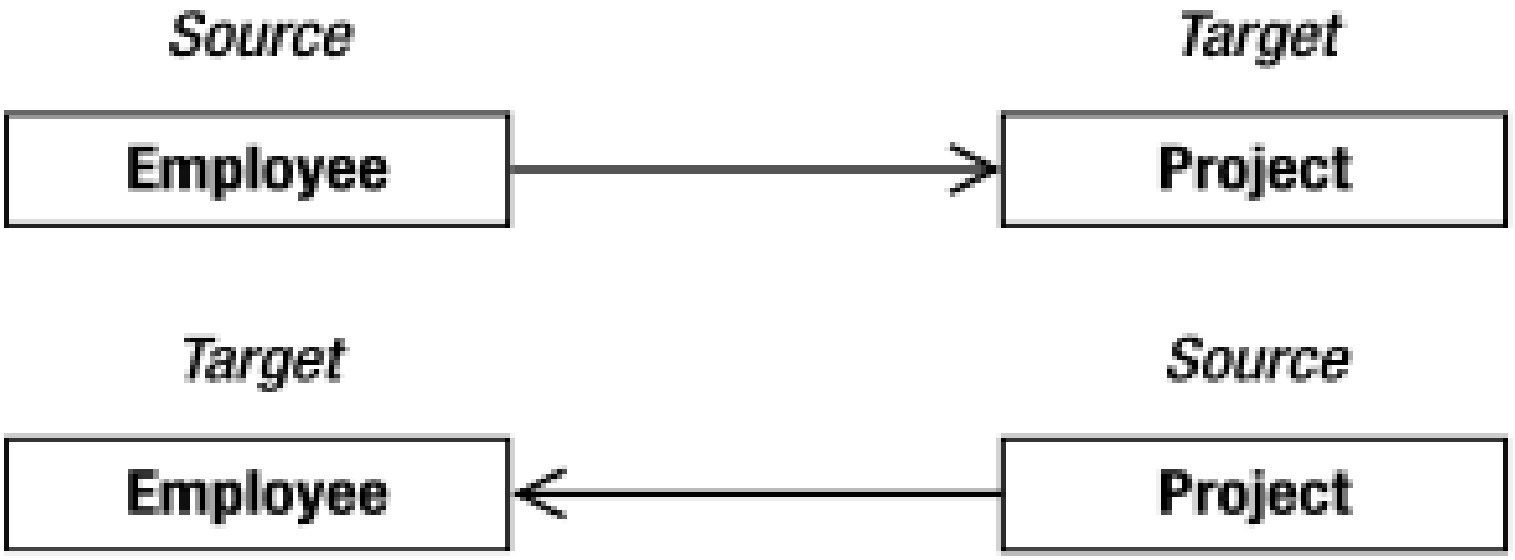
Conceptos en relaciones

- Roles



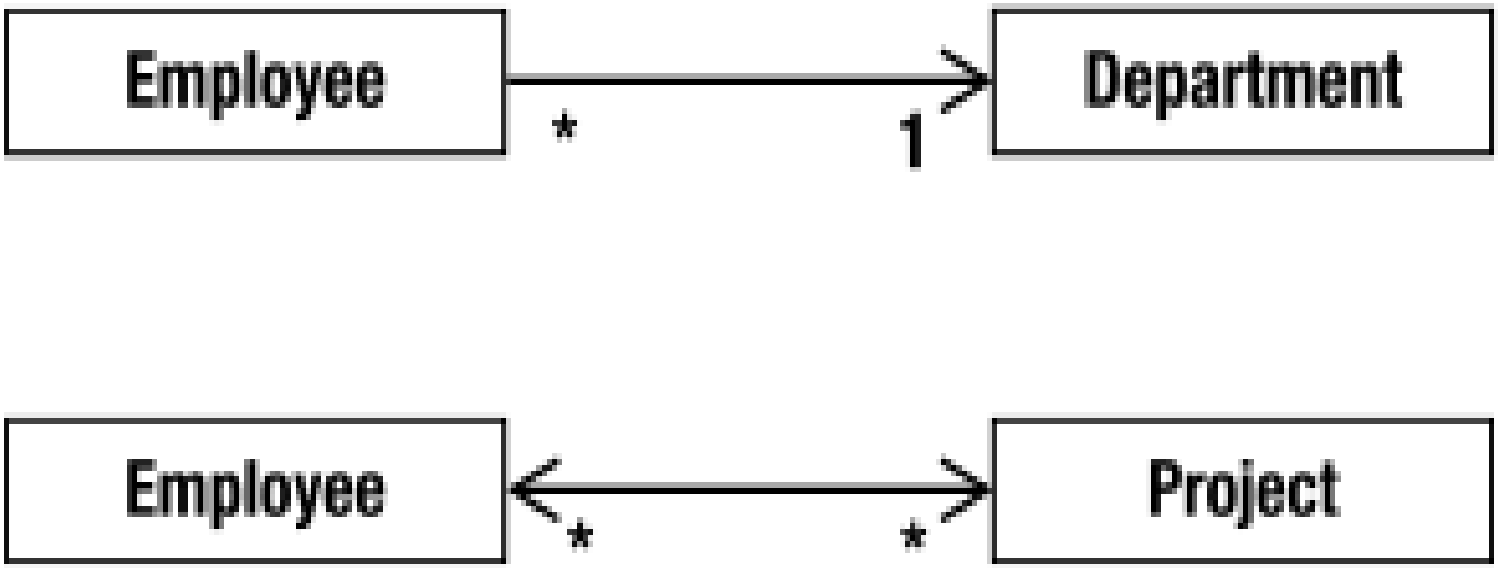
Conceptos en relaciones

- Direccionalidad



Conceptos en relaciones

- Cardinalidad



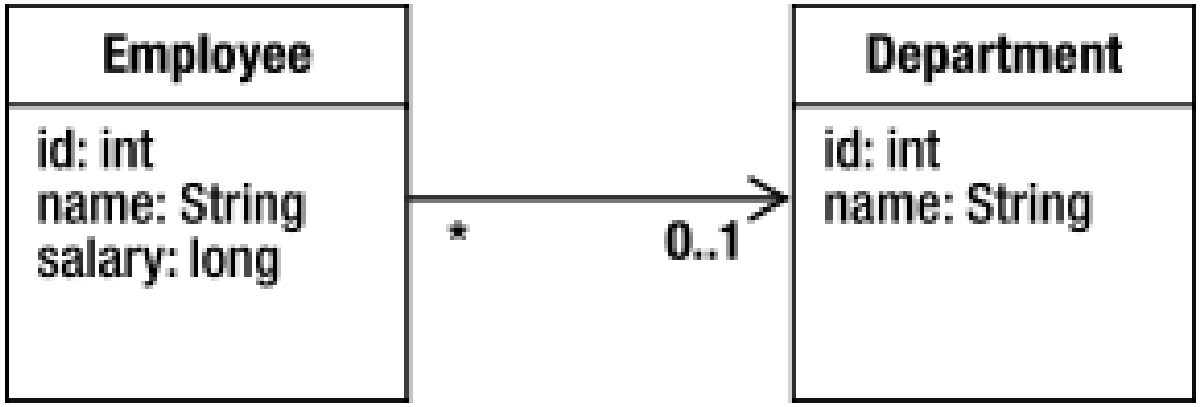
Conceptos en relaciones

- Opcionalidad
  - Permite indicar si un rol esta presente o no
  - Indica si al momento de crear una entidad, la entidad relacionada debe estar presente o no

Conceptos en relaciones

- Tipos de relaciones
  - Many-to-one
  - One-to-one
  - One-to-many
  - Many-to-many

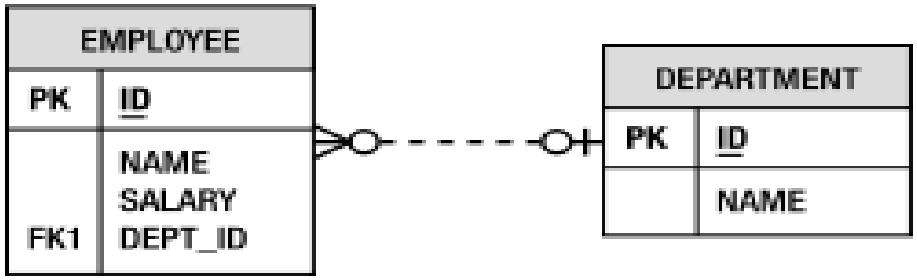
Many-to-one




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

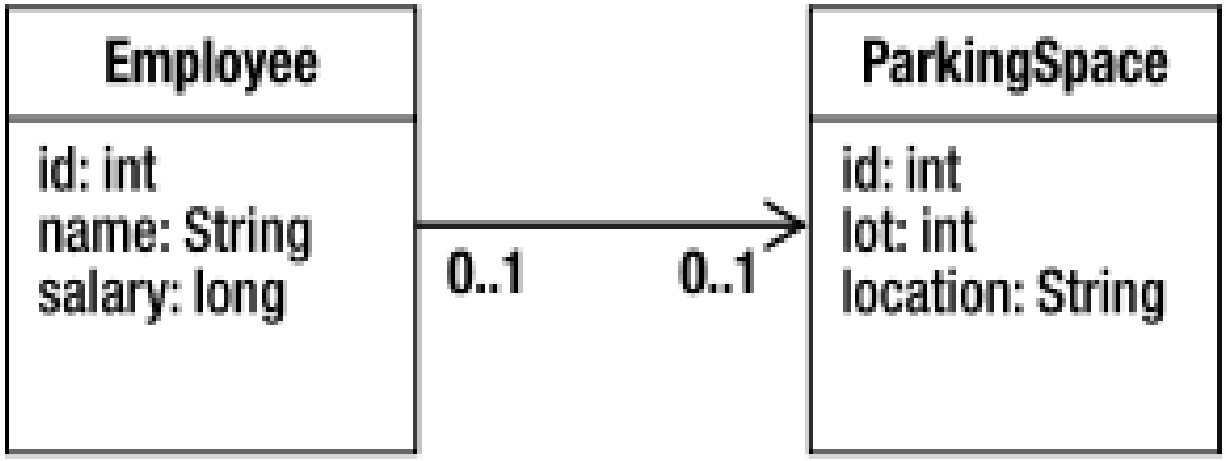
Many-to-one

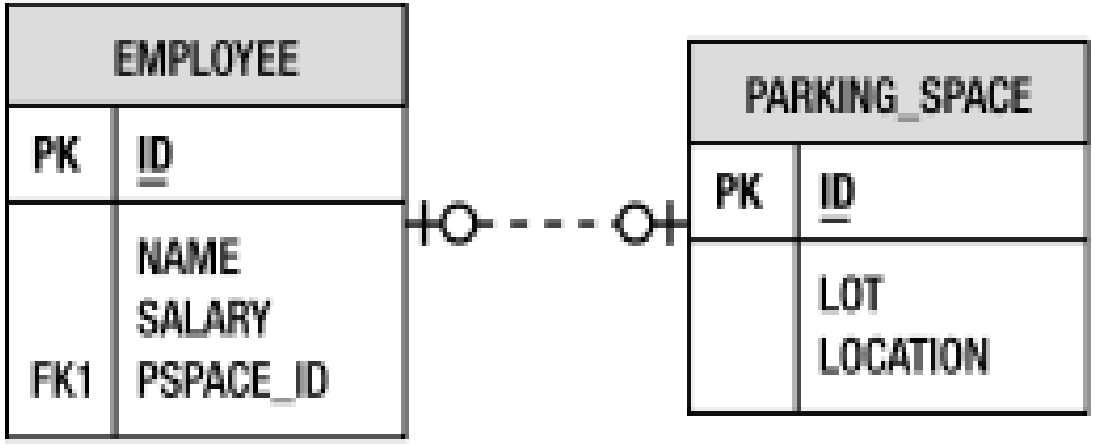
- Para indicar las columnas que permiten realizar el join a nivel de la base de datos, utilizamos la anotacion @JoinColumn




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

One-to-one

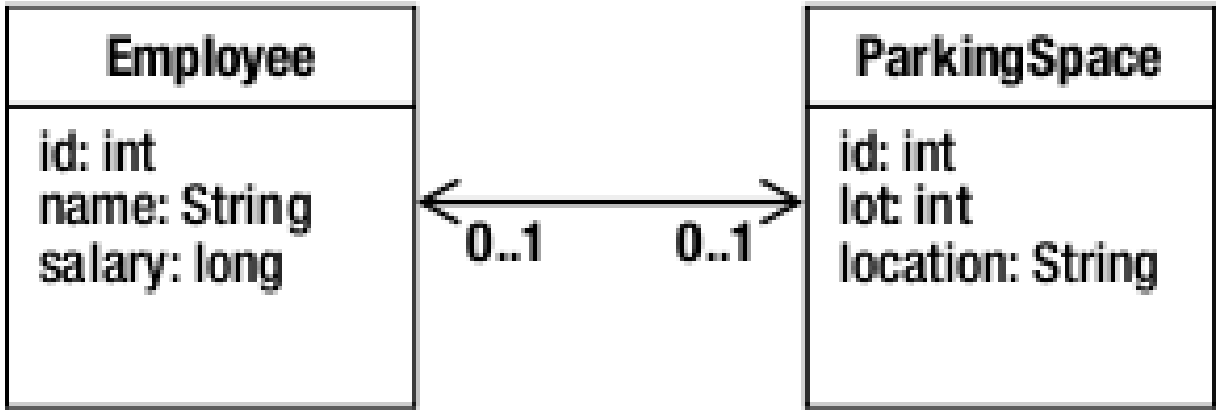

Employee
id: int
name: String
salary: long
0..1
0..1
ParkingSpace
id: int
lot: int
location: String


EMPLOYEE
PK
ID
NAME
SALARY
FK1
PSPACE_ID
PARKING_SPACE
PK
ID
LOT
LOCATION

One-to-one

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

One-to-one bidireccional

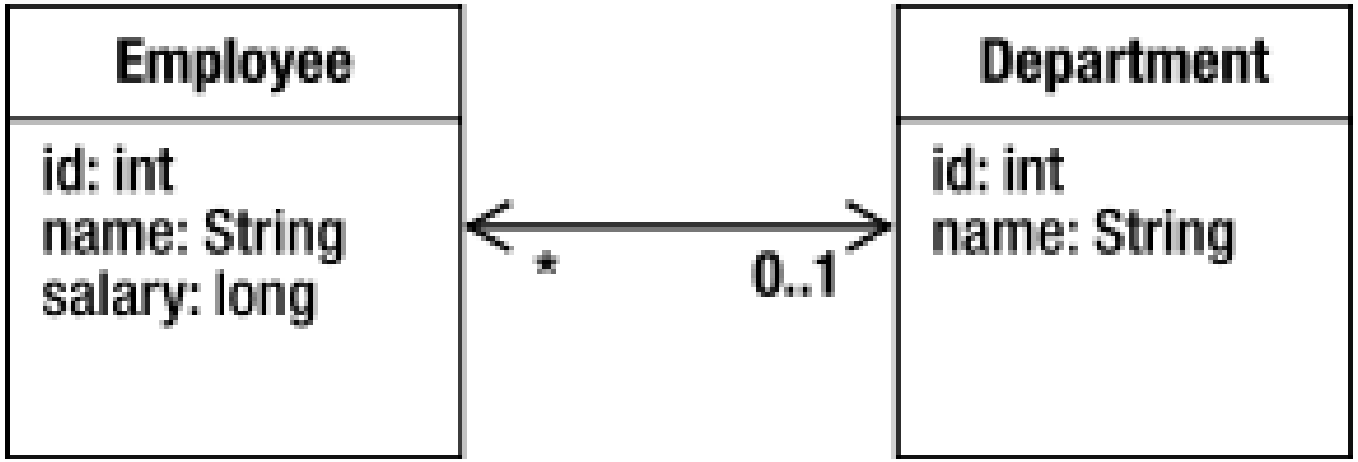



```
@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}
```

mappedBy

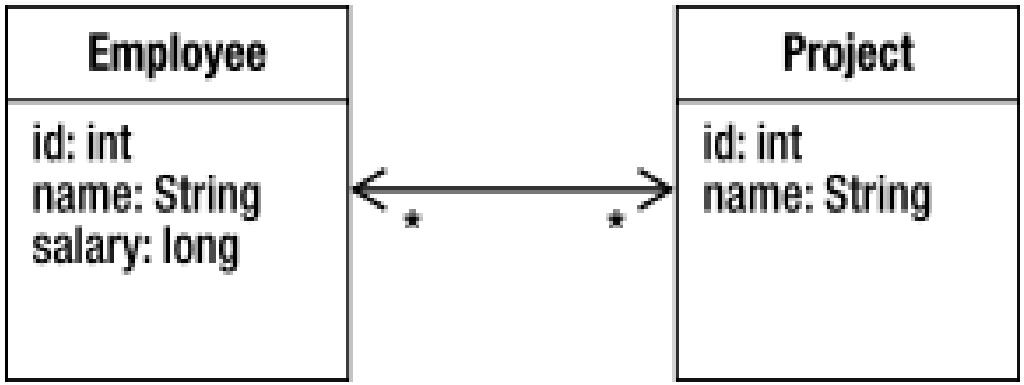
- Este atributo debe ser especificado en la entidad que no define una @JoinColumn
- Esto es lo que se denomina el lado inverso de la relacion
- mappedBy no puede ser especificado de ambos lados de la relacion

One-to-many




```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```

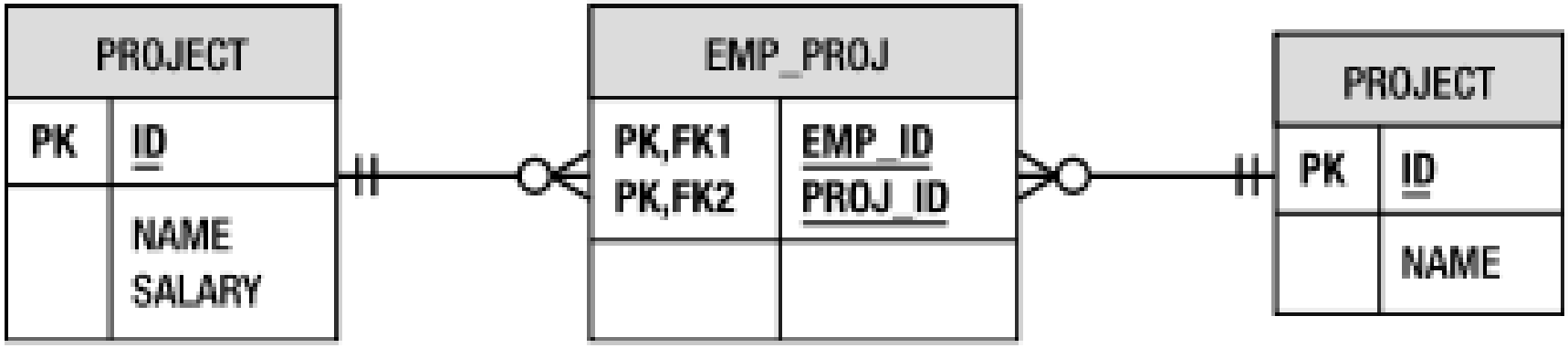
Many-to-many




```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}
```

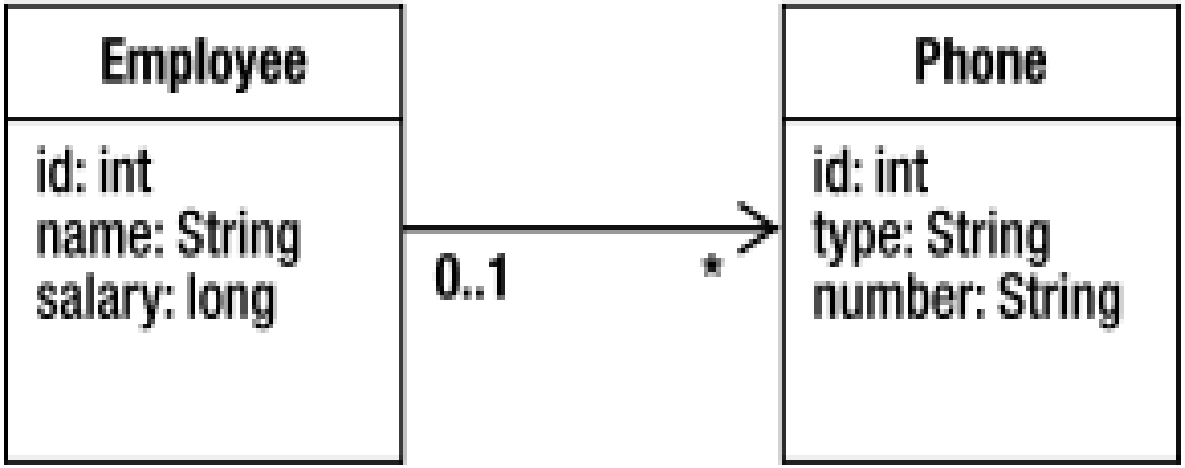
```
@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

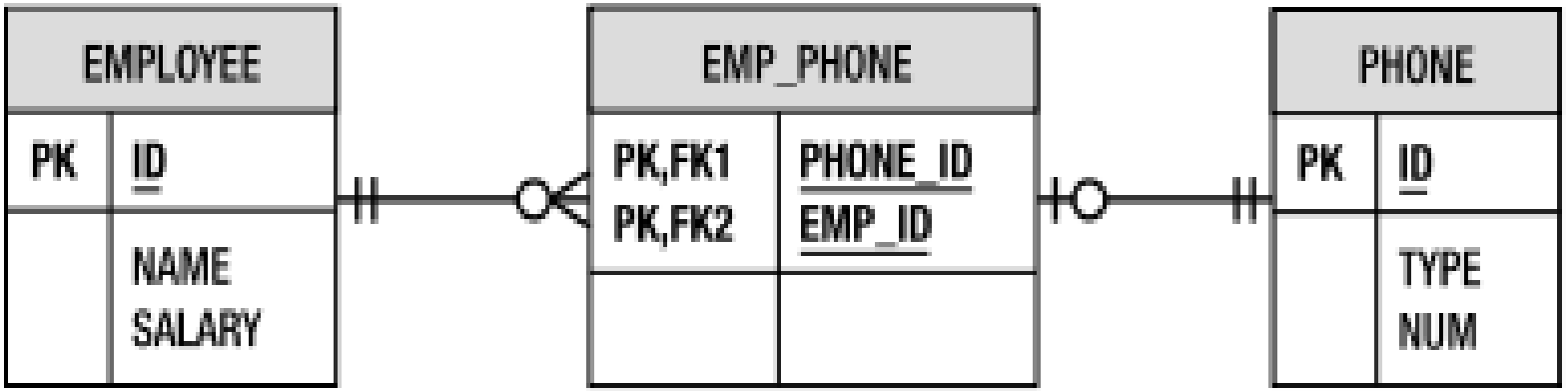
Join Tables




```
@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> projects;
    // ...
}
```

One-to-many unidireccionales





One-to-many unidireccionales

```
@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;
    // ...
}
```

Relaciones lazy

- El “fetch mode” regula la forma en que los objetos participantes de una relacion son cargados a memoria
- Los cuatro tipos de relaciones permiten especificar el “fetch mode”
- Este puede ser EAGER o LAZY
  - Las relaciones monovaluadas son EAGER por defecto
  - Las relaciones multivaluadas son LAZY por defecto

Relaciones lazy

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    // ...
}
```

Queries

- La consulta mas simple que podemos hacer en JPQL, selecciona todas las entidades de un determinado tipo
- La principal diferencia entre SQL y JPQL, es que elegimos entidades del dominio, y no tablas

```
SELECT e
FROM Employee e
```

Queries

- Usando la notación de “.” podemos navegar las relaciones y atributos de una entidad, aun si esta no esta referenciada en el FROM

```
SELECT e.name
FROM Employee e

SELECT e.department
FROM Employee e
```

Queries

- Podemos filtrar los resultados utilizando el WHERE como en SQL
- La diferencia como antes, es que usamos expresiones con entidades, en vez de expresiones con columnas

```
SELECT e
FROM Employee e
WHERE e.department.name = 'NA42' AND
      e.address.state IN ('NY','CA')
```

Queries

- En un SELECT, no podemos devolver colecciones de entidades, siempre debe ser un valor monovaluado
- Podemos realizar JOINs entre entidades para obtener elementos relacionados

```
SELECT p.number
FROM Employee e, Phone p
WHERE e = p.employee AND
      e.department.name = 'NA42' AND
      p.type = 'Cell'
```

Queries

- Si entre dos entidades existe una relación (como el caso anterior), es posible usar el operador JOIN en el FROM
- Este operador realiza el join, pero también provee las condiciones de joineo en forma automática, sin que tengamos que especificarlo

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND
      p.type = 'Cell'
```

Queries

- Se soportan las funciones de agregación presentes siguientes (como en el caso SQL)
  - AVG, COUNT, MIN, MAX, SUM
- Los resultados pueden agruparse utilizando GROUP BY

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

Consultas dinámicas

- Este tipo de consultas puede definirse pasando el string JPQL al método createQuery del entity manager
- No hay restricciones en la definición de la query
- Debemos tener en cuenta, que cada invocación de estas, implica analizar y parsear la consulta
- Si bien en general se usan caches para consultas, podría verse afectado su uso según la forma de utilizar la consulta

Consultas dinámicas

- Este es un ejemplo de consulta dinámica..

```
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();
}
```

Utilizando parámetros

```
private 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();
}
```

Consultas nombradas

- Las named queries son un mecanismo útil para organizar las consultas de una aplicación, así como para aumentar la performance de una aplicación
- Usamos la anotación @NamedQuery, colocándola en cualquier entidad de la aplicación
- En general se coloca la query en la entidad que corresponde mas directamente con la consulta

Consultas nombradas

```
@NamedQuery(name="findSalaryForNameAndDepartment",
            query="SELECT e.salary " +
                  "FROM Employee e " +
                  "WHERE e.department.name = :deptName " +
                  "AND   e.name = :empName")
@Entity
public class Employee { ...
```

Consultas nombradas

- El nombre de la consulta debe ser único en todo el persistence unit
- Es común usar el nombre de la entidad como prefijo para evitar problemas
  - Employee.findAll, en vez de findAll
- A nivel de especificación, no esta definido que ocurre cuando hay un conflicto de nombres
  - Las opciones son un error en deployment, o la sobre escritura de una de las consultas

Consultas nombradas

- Se puede definir mas de una consulta
- Para esto utilizamos la anotación @NamedQueries

```
@NamedQueries({
    @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey",
                query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName",
                query="SELECT e FROM Employee e WHERE e.name = :name")
})
```

Consultas nombradas

- El string de este tipo de consultas no puede ser alterado por el programa
- Cualquier variación en la forma en que la consulta recupera los datos, debe hacerse a través del uso de parámetros
- Para utilizar una de estas consultas, debemos utilizar el método createNamedQuery, del entity manager

Consultas nombradas

```
public class EmployeeServiceBean implements EmployeeService {
    // ...
    EntityManager em = ...;
    // ...
    public Employee findEmployeeByName(String name) {
        return em.createNamedQuery("Employee.findByName",
                                   Employee.class)
                 .setParameter("name", name)
                 .getSingleResult();
    }
    // ...
}
```

Tipos de parámetros

- JPA soporta parámetros nombrados y posicionales
- Usamos el método setParameter() de la interfaz Query o TypedQuery
  - El primer argumento es el nombre o la posicion
  - El segundo parametro es el valor
  - Para el caso de fechas, se necesita un tercer valor, indicando el tipo de fecha: java.sql.Date, java.sql.Time, java.sql.TimeStamp

Tipos de parámetros

```
@NamedQuery(name="findEmployeesAboveSal",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary > :sal")

public List<Employee> findEmployeesAboveSal(Department dept,
                                            long minSal) {
   return em.createNamedQuery("findEmployeesAboveSal",
                              Employee.class)
            .setParameter("dept", dept)
            .setParameter("sal", minSal)
            .getResultList();
}
```

Tipos de parámetros

- Un aspecto interesante de JPQL, es que podemos utilizar como parámetro entidades completas
- En el caso de las fechas, debemos especificar el tipo de argumento temporal utilizado
- Para esto, usamos el enumerado TemporalType

Tipos de parámetros

```
public List<Employee> findEmployeesHiredDuringPeriod(Date start,
                                                     Date end) {
    return em.createQuery("SELECT e " +
                          "FROM Employee e " +
                          "WHERE e.startDate BETWEEN ?1 AND ?2",
                          Employee.class)
             .setParameter(1, start, TemporalType.DATE)
             .setParameter(2, end, TemporalType.DATE)
             .getResultList();
}
```

Tipos de parámetros

- El mismo parámetro puede ser usado múltiples veces

```
@NamedQuery(name="findHighestPaidByDepartment",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary = (SELECT MAX(e.salary) " +
                  "                  FROM Employee e " +
                  "                  WHERE e.department = :dept)")

public Employee findHighestPaidByDepartment(Department dept) {
     return em.createNamedQuery("findHighestPaidByDepartment",
                                Employee.class)
                           .setParameter("dept", dept)
                           .getSingleResult();
}
```

Ejecutando consultas

- Query y TypedQuery proveen tres formas de ejecutar consultas, dependiendo de la cantidad de resultados esperados
- Para consultas que retornan valores, podemos invocar
  - getSingleResult() si se espera un solo resultado
  - getResultList() si se espera mas de un valor
- Para actualizaciones podemos usar executeUpdate()

Ejecutando consultas

```
public void displayProjectEmployees(String projectName) {
    List<Employee> result = em.createQuery(
                            "SELECT e " +
                            "FROM Project p JOIN p.employees e "+
                            "WHERE p.name = ?1 " +
                            "ORDER BY e.name",
                            Employee.class)
                        .setParameter(1, projectName)
                        .getResultList();
    int count = 0;
    for (Employee e : result) {
        System.out.println(++count + ": " + e.getName() + ", " +
                           e.getDepartment().getName());
    }
}
```

Resultados de una consulta

- El tipo resultado de la consulta depende de las expresiones utilizadas en el SELECT
- Por ejemplo, si el resultado del SELECT es de tipo Employee, entonces getResultList() devolverá una lista de Employee
- Algunos posibles resultados
  - Tipos básicos como String o primitivos
  - Tipos de entidad
  - Array de Object
  - Tipos definidos por el usuario, a traves del uso del new

Resultados de una consulta

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery(
                            "SELECT e.name, e.department.name " +
                            "FROM Project p JOIN p.employees e " +
                            "WHERE p.name = ?1 " +
                            "ORDER BY e.name")
                    .setParameter(1, projectName)
                    .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        Object[] values = (Object[]) i.next();
        System.out.println(++count + ": " +
                           values[0] + ", " + values[1]);
    }
}
```

Resultados de una consulta

- Es posible definir una clase utilitaria para almacenar los valores devueltos por la consulta
- Esta clase definida debe poseer un constructor que concuerde en cantidad y tipos con los elementos devueltos por el SELECT
- Para crear una nueva instancia de esta clase, usamos el operador NEW a continuación del SELECT

Bulk delete

```
public void removeEmptyProjects() {
      em.createQuery(
            "DELETE FROM Project p " +
            "WHERE p.employees IS EMPTY")
        .executeUpdate();
}
```

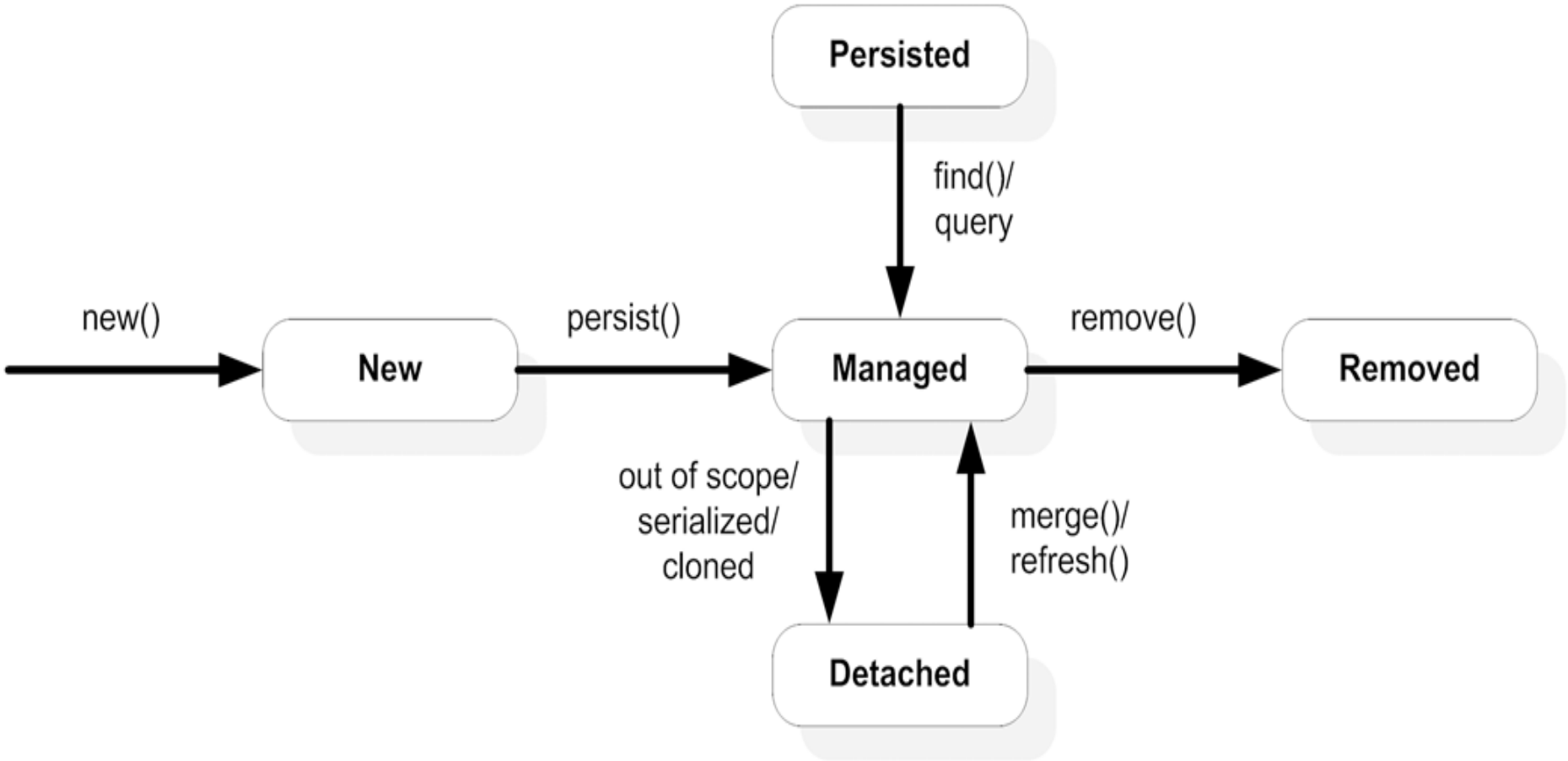
Entity Manager

- Las entidades no se persisten a si mismas cuando son creadas, así como tampoco son removidas cuando el garbage collector recicla los objetos
- Es la lógica de la aplicación la que debe manejar el ciclo de vida de persistencia
- Para este propósito, JPA provee la interfaz EntityManager

Entity Manager

```
public class EmployeeClient {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        List<Employee> emps =
            em.createQuery("SELECT e FROM Employee e").getResultList();
        for (Employee e : emps) {
            System.out.println(e.getId() + ", " + e.getName());
        }
        em.close();
        emf.close();
    }
}
```

Estados de una entidad



Transactions

- A nivel de codigo Java, las operaciones que queramos esten enlistadas en la transaccion deben estar encerradas entre un begin y un commit

```
em.getTransaction().begin();
em.persist(entidad);
em.getTransaction().commit();
```

Operaciones disponibles

- La interfaz EntityManager define una serie de métodos que permiten manejar el ciclo de vida de las entidades de una aplicación
- Estos métodos incluyen operaciones como
  - Persistencia de entidades
  - Recuperación de entidades
  - Obtención de consultas
  - Inicio y fin de transacciones

persist()

- Esta operación acepta una nueva entidad como parámetro y la transforma en **managed**
- Si la entidad a persistir ya esta siendo administrada por el contexto de persistencia, entonces la operación la ignora
- Podemos usar la operación contains() para verificar si la entidad esta **managed**
- La invocación de esta operación no significa que la entidad es persistida en ese momento

persist()

```
Department dept = em.find(Department.class, 30);
Employee emp = new Employee();
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp);
```

find()

- Cuando una entidad debe ser localizada usando su primary key, entonces el método a utilizar es find()
- Esta operación devuelve una entidad que queda en estado managed
  - Salvo que estemos en un transaction-scoped entity manager, sin transacción activada
  - En este caso, el objeto es devuelto sin asociarlo a ningún contexto de persistencia

find()

```
Department dept = em.find(Department.class, 30);
Employee emp = new Employee();
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp);
```

remove()

- Para remover una entidad, debemos pasar al método remove() una instancia de la entidad, que se encuentre en estado managed
- Cuando el estado del contexto de persistencia se sincronice con la base de datos, la entidad sera removida de la misma
- Hay que prestar atención a las relaciones con otras entidades, para evitar problemas de integridad referencial

remove()

- Un ejemplo que implica una relación con otra entidad...

```
Employee emp = em.find(Employee.class, empId);
ParkingSpace ps = emp.getParkingSpace();
emp.setParkingSpace(null);
em.remove(ps);
```

Propagación de operaciones

- Por defecto, las operaciones anteriores solo aplicación sobre las entidades que son pasadas como parámetro a la función
- Para algunas operaciones como remove(), este comportamiento es aceptable
- Pero para otras como persist(), es deseable que si dos objetos relacionados son creados, ambos dos sean persistidos automaticamente

Propagación de operaciones

- Sin propagación, tendríamos que hacer esto...

```
Employee emp = new Employee();
emp.setId(2);
emp.setName("Rob");
Address addr = new Address();
addr.setStreet("645 Stanton Way");
addr.setCity("Manhattan");
addr.setState("NY");
emp.setAddress(addr);
em.persist(addr);
em.persist(emp);
```

Propagación de operaciones

- JPA provee un mecanismo para especificar cuando operaciones como el persist(), deben propagarse a través de una relación
- Esto se denomina **cascade**, y se especifica a nivel de la relación
- El modo de cascade de una relación, se define en el enumerado CascadeType, tomando los valores
  - PERSIST, REFRESH, REMOVE, MERGE, DETACH

Cascade persist

```
@Entity
public class Employee {
    // ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
    // ...
}
```

Cascade persist

- Para que todo funcione correctamente, la aplicación debe asegurarse que Address este asociado a Employee antes de que se invoque persist() sobre la instancia de Employee
- Esto permite que el código se desentienda de la operación persist() para la instancia de Address
- Las definiciones del cascade de la relación, se definen en forma **UNIDIRECCIONAL**

Cascade remove

- Es similar al anterior, permitiendo remover un grafo de objetos, operando en la raíz del grafo

```
@Entity
public class Employee {
    // ...
    @OneToOne(cascade={CascadeType.PERSIST,
                       CascadeType.REMOVE})
    ParkingSpace parkingSpace;
    @OneToMany(mappedBy="employee",
               cascade={CascadeType.PERSIST,
                        CascadeType.REMOVE})
    Collection<Phone> phones;
    // ...
}
```

clear()

- Esta operación permite limpiar el contexto de persistencia de las entidades que han quedado en estado managed, asociadas al mismo
- Este contexto no es limpiado automáticamente cuando una transacción es confirmada
- Por este motivo, puede ser necesario realizar una limpieza explicita en algún momento de la ejecución del programa

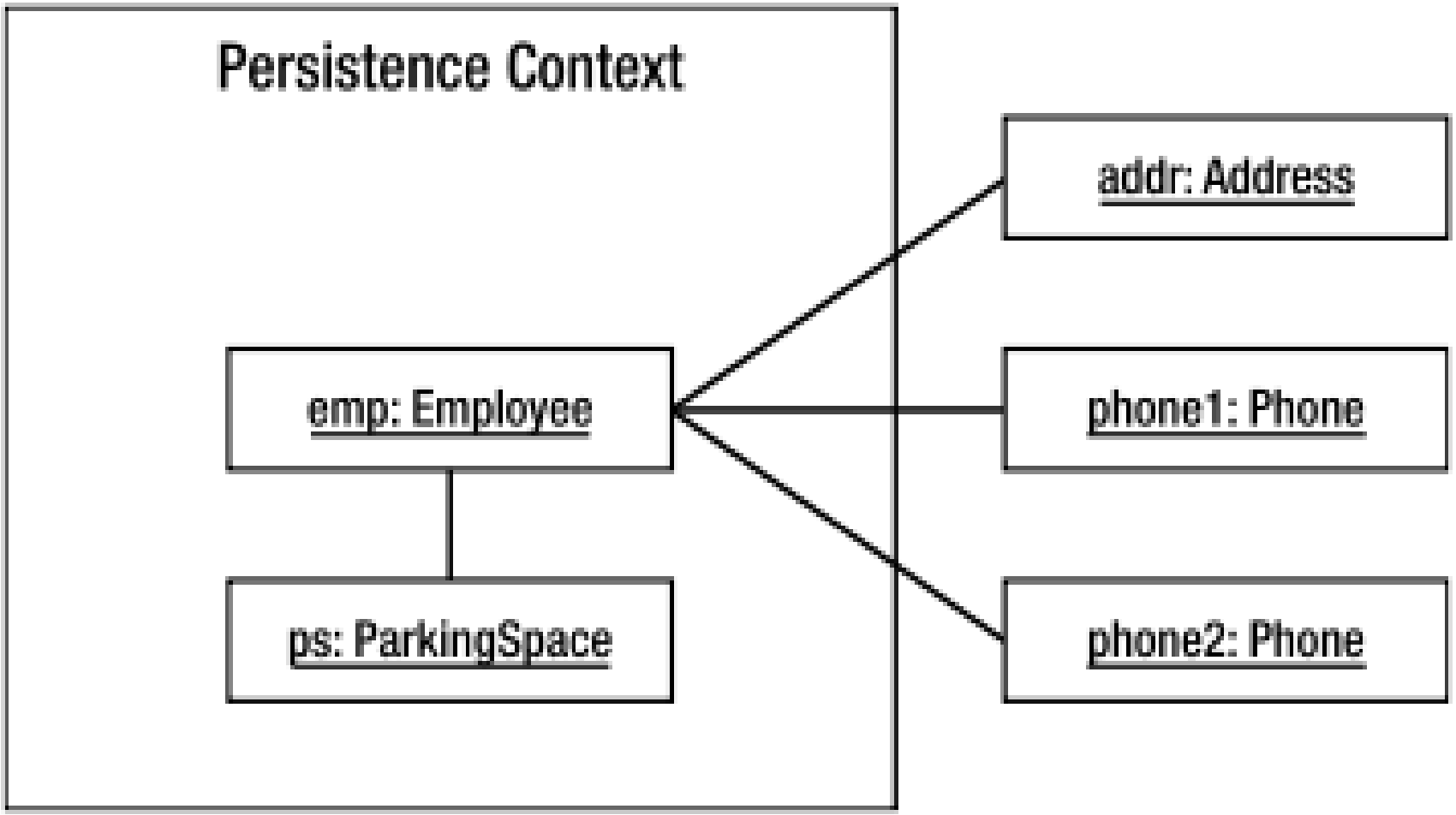
Sincronización con la base

- Cada vez que el entity manager genera SQL y ejecuta el mismo contra la base de datos, decimos que el contexto de persistencia ha sido "flusheado" (**flush**)
- Un flush esta garantizado en dos situaciones:
  - Cuando la transacción es confirmada
  - Cuando la operación flush() del entity manager es invocada

flush()

- Un flush esta compuesto por tres elementos:
  - Entidades nuevas que deben ser persistidas
  - Entidades modificadas que deben ser actualizadas
  - Entidades removidas que deben ser eliminadas de la base de datos
- Toda esta información es administrada por el contexto de persistencia, manteniendo enlaces a las diferentes entidades de las categorías anteriores

flush()


Persistence Context
addr: Address
emp: Employee
phone1: Phone
ps: ParkingSpace
phone2: Phone

Detachment y Merging

- Una entidad se encuentra en estado detached, cuando ya no esta asociada a un persistence context
- La entidad estuvo en algún momento managed, pero el contexto de persistencia termino o la entidad fue transformada, de forma de perder este vinculo con el contexto de persistencia
- El contexto de persistencia ya no trackea los cambios sobre esta entidad

Detachment y Merging

- El opuesto del detachment es el merge
- El proceso de merge ocurre cuando el entity manager integra el estado de una entidad detached, dentro del contexto de persistencia
- El merge permite que una entidad sea modificada “offline”, incorporando todos esos cambios “de golpe” en el contexto de persistencia
- Estos cambios se almacenar al confirmar la transaccion

Detachment

- Una entidad puede pasar a estado detached, cuando:
  - La transacción asociada al entity manager termina. Las entidades asociadas al persistence context, quedan detached
  - Un entity manager es cerrado
  - Se invoca el método clear() del entity manager
  - Se invoca el método detach() para una entidad
  - La transacción asociada al entity manager es cancelada
  - Cuando una entidad es serializada

Detachment

- Las entidades detached, presentan una complicacion a la hora de trabajar con atributos y relaciones LAZY
- Por ejemplo...

```
@Entity
public class Employee {
    // ...
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    // ...
}
```

Detachment

- En el caso anterior, si el Employee es levantado de la base, la entidad ParkingSpace sera cargada en demanda
- Si la entidad Employee se vuelve detached antes de cargar la entidad ParkingSpace, entonces la relación no puede resolverse automaticamente
- En este ultimo caso, se produce una excepcion LazyInitializationException...

Merging

- La operación merge() permite colocar el estado de una entidad en el persistence context
- Por ejemplo...

```
public void updateEmployee(Employee emp) {
    em.merge(emp);
    emp.setLastAccessTime(new Date());
}
```

Merging

- Existe una sutileza con el manejo del merge()
- El argumento a merge() no pasa a estado managed luego de la invocación
- Por este motivo, el ejemplo anterior es incorrecto, ya que los cambios a la fecha de ultimo acceso no seran registrados
- La operación merge() devuelve como resultado una instancia diferente de la entidad, la cual si esta en estado managed

Merging

- La entidad retornada por merge(), es la entidad colocada en el persistence context por la operación
- Esta entidad contendra el estado que se encontraba en la entidad pasada como parametro a merge()
- Para operar correctamente, entonces debemos hacerlo sobre el resultado de la operacion

Merging

- El ejemplo correcto seria...

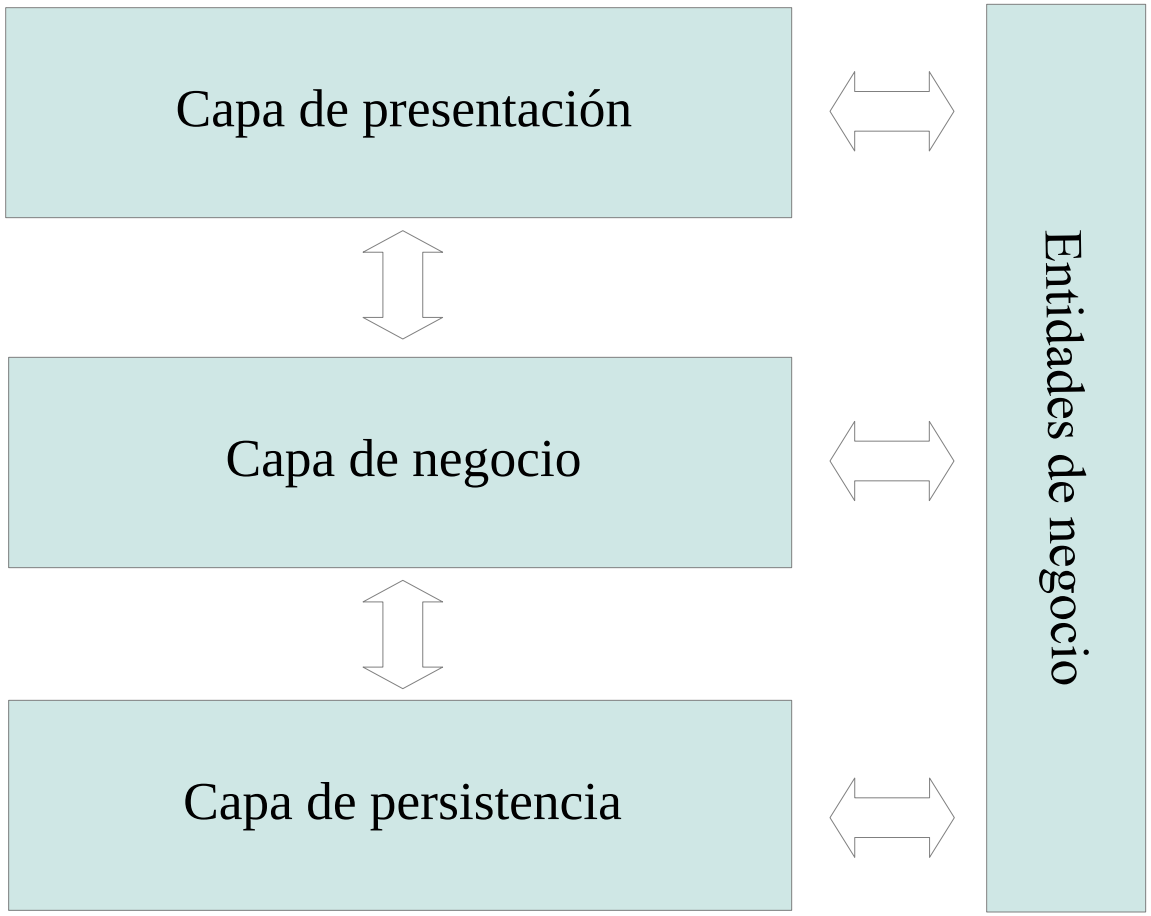
```
public void updateEmployee(Employee emp) {
    Employee managedEmp = em.merge(emp);
    managedEmp.setLastAccessTime(new Date());
}
```

Estilos arquitectónicos

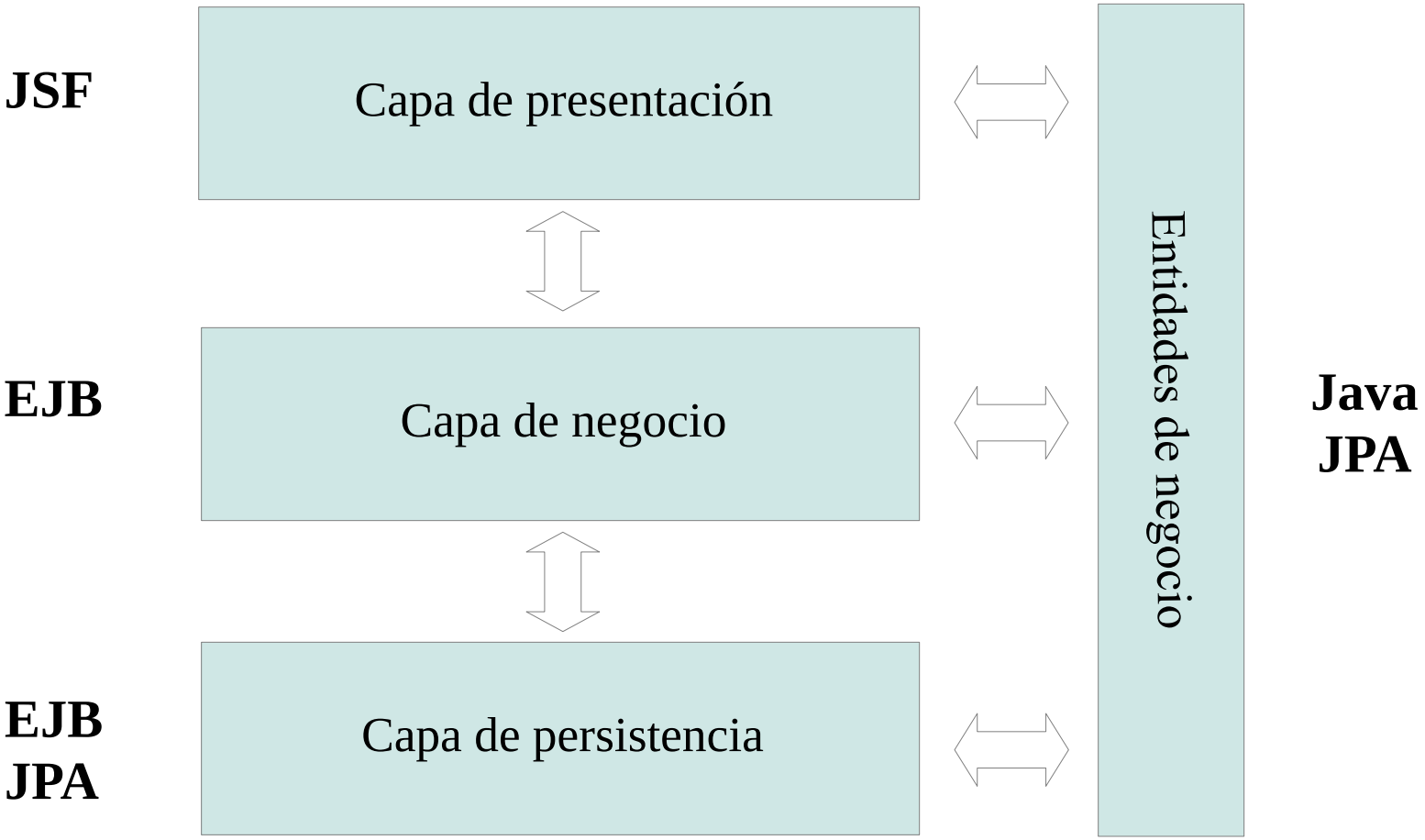
Estilos arquitectónicos

- Existen múltiples formas de combinar las tecnologías antes presentadas
  - Java Server Faces
  - Enterprise Java Beans
  - Java Persistence API
- Veamos algunas formas simples de combinarlas para obtener sistemas funcionales

Arquitectura típica

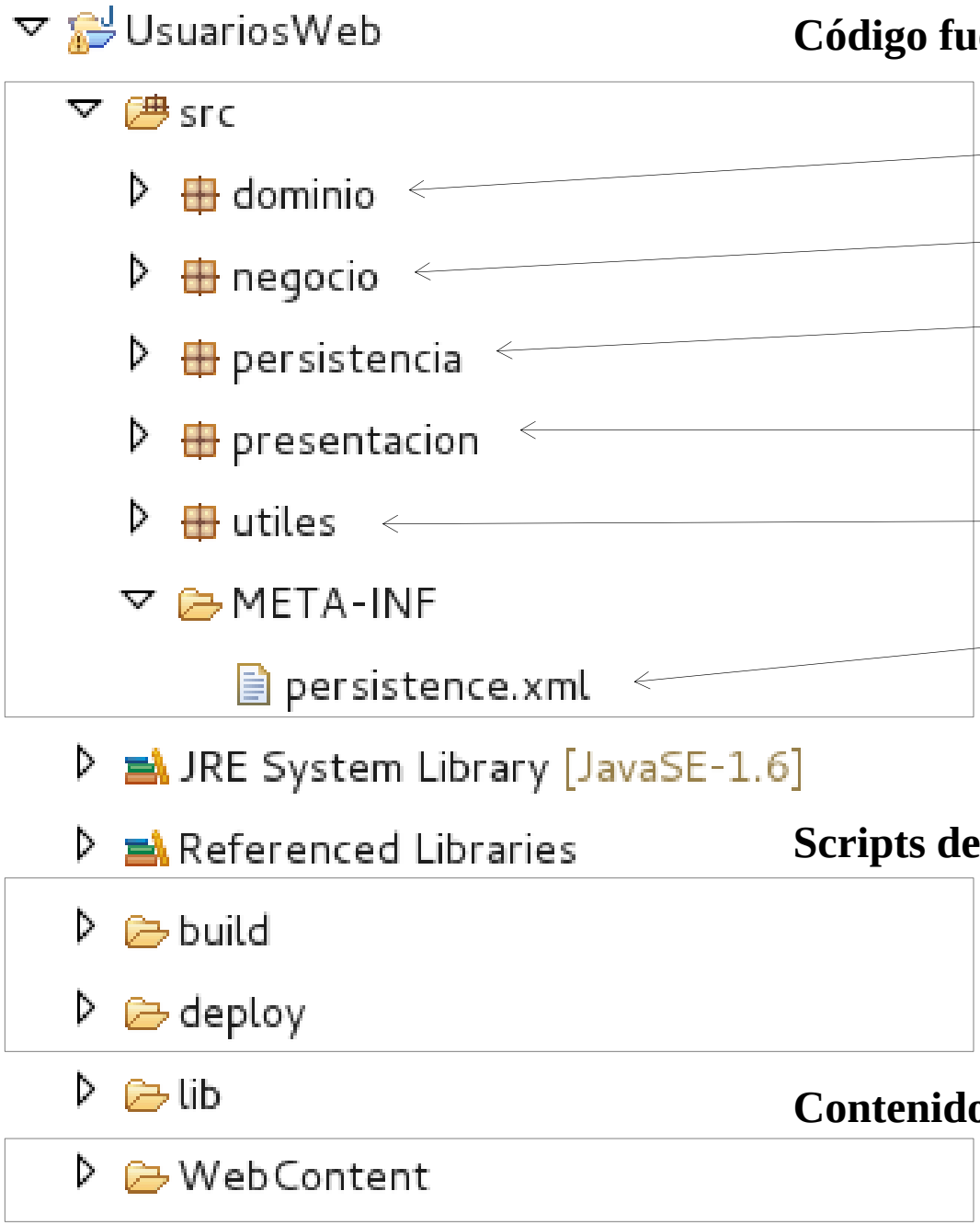


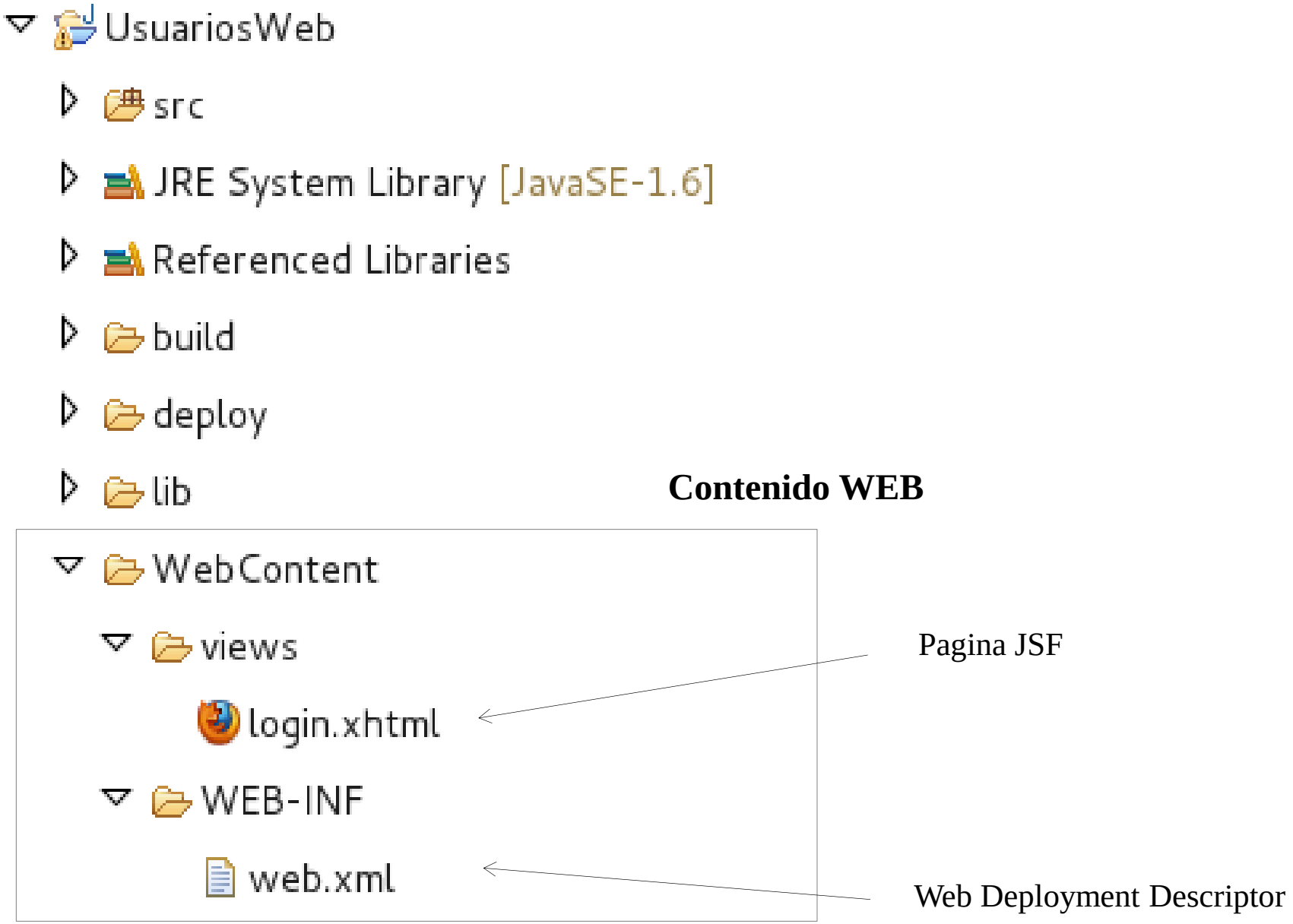
Arquitectura típica



UsuariosWeb (Aplicación de ejemplo)

- Es una aplicación sencilla, desarrollada utilizando las APIs antes presentadas
- Es un proyecto WEB que incorpora componentes EJB, entidades JPA y paginas JSF
- Puede ser desplegada dentro de un Tomcat o un Jboss
- Veremos a continuación las diferentes partes que componen la misma


UsuariosWeb
Código fuente de la aplicación
src
dominio
negocio
persistencia
presentacion
utiles
META-INF
persistence.xml
Entidades de dominio
Componentes EJB negocio
Componentes EJB persistencia
Managed Beans JSF
Clases con utilidades varias
Configuración de la persistencia JPA
JRE System Library [JavaSE-1.6]
Referenced Libraries
Scripts de building y deployment
build
deploy
lib
Contenido WEB (xhtml, css, imagenes)
WebContent


UsuariosWeb
src
JRE System Library [JavaSE-1.6]
Referenced Libraries
build
deploy
lib
Contenido WEB
WebContent
views
login.xhtml
WEB-INF
web.xml
Pagina JSF
Web Deployment Descriptor

```java
package dominio;

import java.io.Serializable;

@Entity
@Table(name="TBL_USUARIO" )
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY )
    @Column(name="ID" ,nullable=false )
    private Integer id;

    @Column(name="LOGIN" ,length=100 ,nullable=false )
    private String login = "";

    @Column(name="PASSWORD" ,nullable=false )
    private String password = "";


    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getLogin() {
        return login;
    }
```

Mappings JPA

Acceso a datos: DAO

```java
package persistencia;

import java.util.List;

@Local
public interface UsuarioDAO {

    public Usuario insert(Usuario entity );

    public void update(Usuario entity );

    public void delete(Usuario entity );

    public void delete(Integer id );

    public Usuario findById(Integer id );

    public List<Usuario> findAll();

    public List<Usuario> buscarUsuario(String login ,String password );

}
```

DAO implementado como EJB
Aprovecha manejo transaccional e inyección de dependencias como JPA

```java
package persistencia;

import java.util.List;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class UsuarioDAOImpl implements UsuarioDAO {

    @javax.persistence.PersistenceContext(unitName="USUARIOS_UNIT" )
    private javax.persistence.EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Usuario insert(Usuario entity )
    {
        try
        {
            em.persist(entity );
            return entity;
        }
        catch (Throwable ex)
        {
            throw ExceptionManager.process(ex);
        }

    }
```

Unidad de persistencia definida en archivo persistence.xml e inyectada por el servidor de aplicaciones

Persistencia de la entidad
JPA queda encapsulado en este método

Nombre de la unidad de persistencia
Debe ser referenciada desde el código Java

Tipo de transacción manejada
por el servidor de aplicaciones

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http:

    <persistence-unit name="USUARIOS_UNIT" transaction-type="JTA">
        <jta-data-source>java:/usuariosds</jta-data-source>
        <class>dominio.Usuario</class>
        <properties>
            <property name="hibernate.hbm2ddl.auto"
                      value="update"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
        </properties>
    </persistence-unit>

</persistence>
```

Nombre del datasource en donde esta definida
la conexión a la base de datos
Es administrada por el servidor de aplicacione

Solo esta entidad sera considerada
con esta conexión a la base de datos

Sabemos que estamos con la implementación de Hibernate
porque hacemos el deploy dentro de JBoss

Interfaz local, acceso dentro del servidor de aplicaciones

```java
package negocio;

import javax.ejb.Local;

@Local
public interface ServiciosSeguridad {

    public Boolean existeUsuario(String login ,String password );

}
```

Función de negocio expuesta a presentación y otros componentes

Componente EJB con manejo transaccional declarativo

```java
package negocio;

import java.util.List;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class ServiciosSeguridadImpl implements ServiciosSeguridad {

    @EJB
    private UsuarioDAO usuarioDAO;

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Boolean existeUsuario(String login ,String password )
    {
        try
        {
            List<Usuario> usuarios = this.usuarioDAO.buscarUsuario(login, password);
            return usuarios != null && usuarios.size() > 0;
        }
        catch (Throwable ex)
        {
            throw ExceptionManager.process(ex);
        }

    }

}
```

Inyección del componente EJB que implementa el DAO

```java
package presentacion;

import java.io.Serializable;

@ManagedBean(name="loginBean" )
@ViewScoped
public class LoginBean implements Serializable {

    private static final long serialVersionUID = 1L;

    @EJB
    private ServiciosSeguridad servicioSeguridad;

    private String txtLogin;
    private String txtPassword;

    public String getTxtLogin() {
        return txtLogin;
    }
    public void setTxtLogin(String txtLogin) {
        this.txtLogin = txtLogin;
    }
    public String getTxtPassword() {
        return txtPassword;
    }
    public void setTxtPassword(String txtPassword) {
        this.txtPassword = txtPassword;
    }

    public String ingresar_ON_CLICK() {
        if (servicioSeguridad.existeUsuario(this.txtLogin, this.txtPassword)) {
            this.showInformationMessage("Credenciales correctas" );
        } else {
            this.showErrorMessage("Credenciales incorrectas" );
        }
        return null;
    }
```

Managed Bean en ViewScope

Inyección de dependencias del componente de negocio

Handler del evento ON_CLICK

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:h="http://java.sun.com/jsf/html">

    <f:view contentType="text/html">
        <h:head>
            <meta content="text/html; charset=utf-8" http-equiv="Content-Type"/>
            <title>Ingreso al sistema</title>
        </h:head>

        <h:body>
            <h:form prependId="false">
                <h:outputText escape="false" value="Login"/>
                <h:inputText value="#{loginBean.txtLogin}" />

                <h:outputText escape="false" value="Password"/>
                <h:inputText value="#{loginBean.txtPassword}" />

                <h:commandButton type="submit"
                                 value="Ingresar"
                                 action="#{loginBean.ingresar_ON_CLICK}"/>
            </h:form>
        </h:body>

    </f:view>
</html>
```

Binding de datos (propiedades)

Binding de métodos