

# Tema 3: Tecnologías para el desarrollo en el servidor

**Servlets y Java Server Pages (JSP)**

**Java Persistence API (JPA)**

**Java Server Faces (JSF)**

**Enterprise Java Beans (EJB)**

Sistemas de Información para Internet  
3º del Grado de Ingeniería Informática (tres menciones)

Departamento de Lenguajes y Ciencias de la Computación  
Escuela Técnica Superior de Ingeniería Informática  
Universidad de Málaga

# **Java Persistence API (JPA)**

**Java  
Persistence**



# Java Persistence API

## Motivación

- Los datos persistentes de una aplicación se suelen almacenar en **bases de datos relacionales**
- Las **relaciones** (tablas) están formadas por “filas” de datos, cada una contiene una colección de valores para cada “columna”
- En los lenguajes **Orientados a Objetos** tenemos **objetos**
- Puede resultar más sencillo para el programador manipular y pensar en términos de objetos

# Java Persistence API

## Motivación

- Para reducir la diferencia entre ambos mundos apareció el ORM (*Object-Relational Mapping*)
- Los *frameworks* de ORM “mapean” las filas de las tablas en objetos (*entidades*) para que puedan ser manipuladas como tales en la aplicación
- Algunos frameworks de ORM: Entity Framework (.NET), Core Data (Cocoa), Doctrine (PHP), Hibernate (Java), EclipseLink (Java), ...

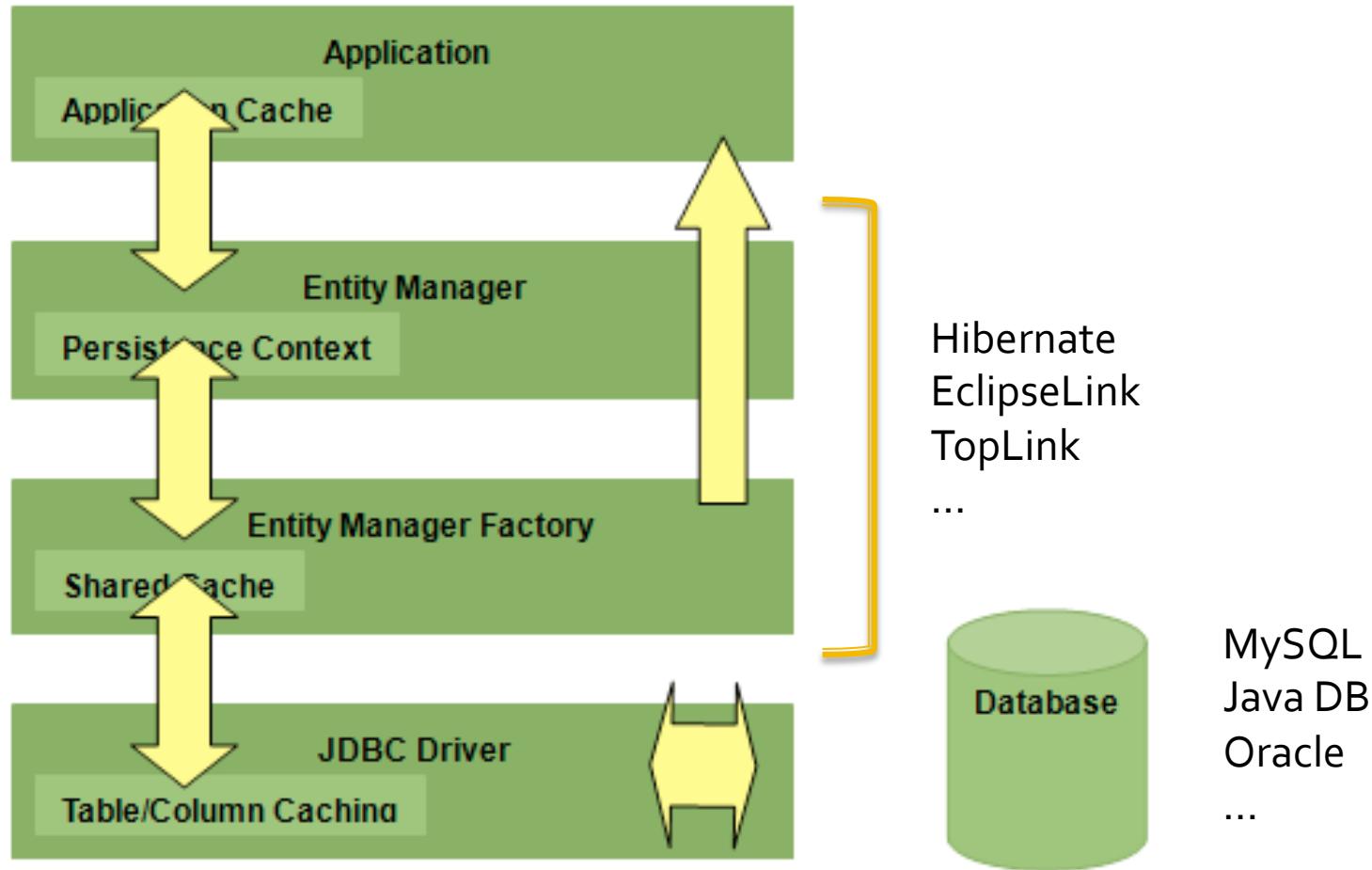
# Java Persistence API

## ¿Qué es?

- Java Persistence API (JPA) es una especificación de ORM para Java
- Para usarlo necesitamos una implementación de JPA
- Algunas implementaciones son: Hibernate, EclipseLink, TopLink, DataNucleus, openJPA, etc...
- EclipseLink es una implementación de referencia

# Java Persistence API

## ¿Qué es?



# Java Persistence API

## ¿Cómo se usa?

- En primer lugar hay que especificar cómo se mapean las clases de entidad

```
@Entity  
public class Producto implements Serializable {
```

- Las instancias de las clases entidad se denominan “entidades” y son objetos que “persisten” en la BBDD
- Las modificaciones en dichos objetos dan lugar a cambios en la BBDD

JPQL

```
List<Producto> lp = em.createQuery("SELECT p FROM Producto p").getResultList();  
  
tx.begin();  
Producto p = em.find(Producto.class, 1234L);  
p.setNombre("pan");  
tx.commit();
```

# Java Persistence API

## Mapeo: clase entidad

- Una clase entidad es un POJO (*Plain Old Java Object*) con anotaciones
- `@Entity` identifica las clases entidad
- `@Id` identifica la clave primaria

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    public Book() {
    }

    // Getters, setters
}
```

Los valores para la clave primaria son generados automáticamente por el motor de persistencia

# Java Persistence API

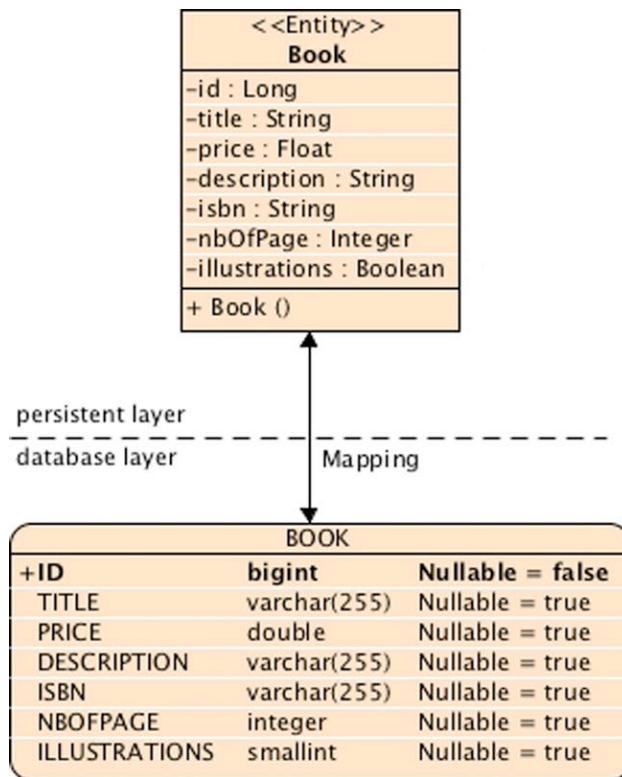
## Mapeo: clase entidad

- Una clase entidad debe cumplir las siguientes características
  - Debe tener un constructor sin argumento
  - Debe ser una clase de primer nivel (*top-level*)
  - No debe ser *final* ni debe tener miembros *final*
  - Conviene que implemente la interfaz **Serializable**

# Java Persistence API

## Mapeo: clase entidad

- El motor de persistencia mapeará la clase entidad a una tabla y cada atributo de dicha clase a una columna de la tabla



```
CREATE TABLE BOOK (
    ID BIGINT NOT NULL,
    TITLE VARCHAR(255),
    PRICE FLOAT,
    DESCRIPTION VARCHAR(255),
    ISBN VARCHAR(255),
    NBOFPAGE INTEGER,
    ILLUSTRATIONS SMALLINT DEFAULT 0,
    PRIMARY KEY (ID)
)
```

# Java Persistence API

## Mapeo: clase entidad

- Podemos ver el esquema de la BBDD que genera el motor de persistencia usando las propiedades

```
javax.persistence.schema-generation.scripts.action create  
javax.persistence.schema-generation.scripts.crea... /tmp/create-traza.ddl
```

- Estas propiedades se encuentran en el fichero de configuración `persistence.xml`
- En el mismo fichero conviene indicar las clases entidad (aunque no siempre es necesario)
- `Netbeans` mantiene este fichero, facilitando la labor de configuración



```
<?> version="1.0" encoding="UTF-8"  
▼ <> persistence version="2.1", xmlns="http://x...  
  ▼ <> persistence-unit name="JPAAppPU", transacti...  
    <> provider (org.eclipse.persistence.jpa.Pe...  
    <> class (jpaapp.modelo.Traza)  
    <> class (jpaapp.modelo.Producto)  
    <> class (jpaapp.modelo.Ingrediente)  
  ▶ <> properties
```

Demo

# Java Persistence API

## Mapeo: clase entidad

- Por defecto el nombre de la tabla coincide con el de la clase entidad
- Podemos cambiar el nombre por defecto usando la anotación `@Table`

```
@Entity
@Table(name = "t_book")
public class Book {

    @Id
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

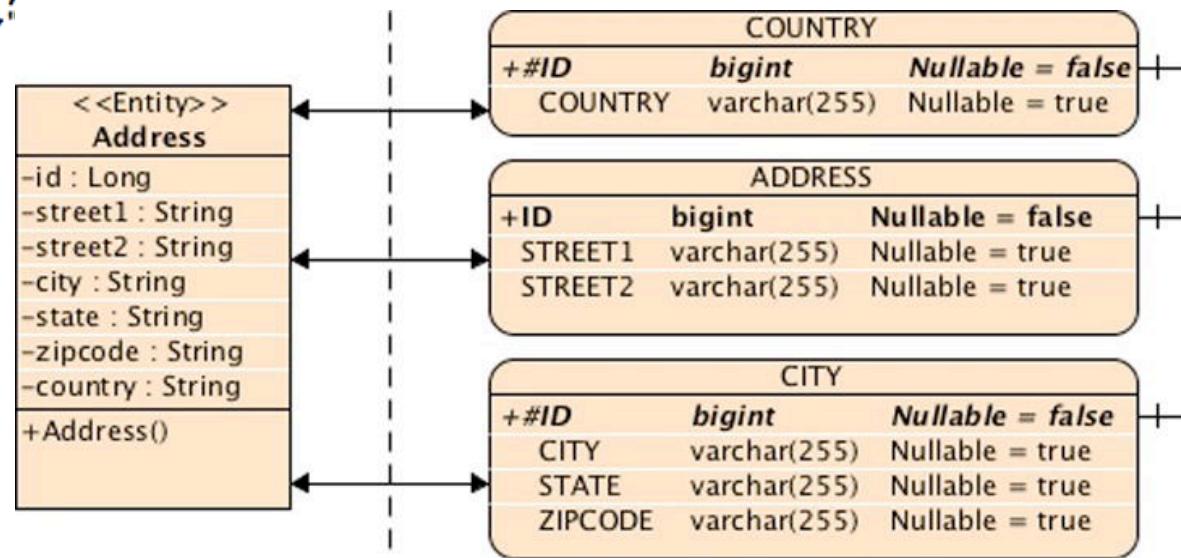
    // Constructors, getters, setters
}
```

# Java Persistence API

## Mapeo: clase entidad

- Podemos separar los atributos de una entidad en distintas tablas con `@SecondaryTables`

```
@Entity  
@SecondaryTables({  
    @SecondaryTable(name = "city"),  
    @SecondaryTable(name = "country")  
})  
public class Address {  
  
    @Id  
    private Long id;  
    private String street1;  
    private String street2;  
    @Column(table = "city")  
    private String city;  
    @Column(table = "city")  
    private String state;  
    @Column(table = "city")  
    private String zipcode;  
    @Column(table = "country")  
    private String country;  
  
    // Constructors, getters, setters  
}
```



# Java Persistence API

## Mapeo: clave primaria

- Si la clave es simple (un solo atributo) se anota con `@Id`
- Opcionalmente podemos indicar que los valores se deben generar automáticamente

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

- Las opciones de generación automática son:
  - SEQUENCE: usa una secuencia (si la BBDD subyacente lo tiene)
  - IDENTITY: usa columnas de ID (si la BBDD lo tiene)
  - TABLE: genera una tabla para mantener secuencias
  - AUTO: deja la decisión en manos del proveedor de persistencia

# Java Persistence API

## Mapeo: clave primaria

- Si la clave es compuesta (varios atributos) tenemos dos formas señalarlos: `@EmbeddedId` y `@IdClass`
  - `@EmbeddedId`: creamos una clase `@Embeddable` que contenga los atributos de la clave primaria

```
@Embeddable
public class NewsId {

    private String title;
    private String language;

    // Constructors, getters, setters, equals, and hashCode
}
```

- Incluimos un atributo de dicha clase en la clase entidad

```
@Entity
public class News {

    @EmbeddedId
    private NewsId id;
    private String content;

    // Constructors, getters, setters
}
```

Opción preferida

# Java Persistence API

## Mapeo: clave primaria

- Si la clave es compuesta (varios atributos) tenemos dos formas señalarlos: `@EmbeddedId` y `@IdClass`
  - `@IdClass`: creamos una clase que contenga los atributos de la clave primaria

```
public class NewsId {  
  
    private String title;  
    private String language;  
  
    // Constructors, getters, setters, equals , and hashCode  
}
```

- Incluimos los atributos de la clave primaria en la clase entidad anotándolos con `@Id` e indicamos la clase de identidad

La clase identidad debe definir `equals` y `hashCode`

```
@Entity  
@IdClass(NewsId.class)  
public class News {  
  
    @Id private String title;  
    @Id private String language;  
    private String content;  
  
    // Constructors, getters, setters  
}
```

Los objetos de identidad se usan para identificar las entidades

# Java Persistence API

## Mapeo: atributos

- Por defecto el nombre del atributo en la tabla coincidirá con el nombre del atributo en la clase
- Podemos cambiar el nombre usando la anotación `@Column`
- Otros parámetros que pueden modificarse con `@Column` son el tamaño de los atributos de texto o la precisión de los atributos numéricos

```
@Entity
public class Book {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "book_title", nullable = false, updatable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    @Column(name = "nb_of_page", nullable = false)
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

# Java Persistence API

## Mapeo: atributos

- Los atributos que almacenen valores temporales pueden contener fechas (DATE), horas (TIME) y marcas de tiempo (TIMESTAMP)
- Pueden ser de los siguientes tipos:
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - java.util.Date
  - java.util.Calendar
- Para los dos últimos es necesario especificar el tipo del valor que contendrá con `@Temporal`

```
private String phoneNumber;  
@Temporal(TemporalType.DATE)  
private Date dateOfBirth;  
@Temporal(TemporalType.TIMESTAMP)  
private Date creationDate;
```

# Java Persistence API

## Mapeo: atributos

- Los tipos enumerados se mapean como enteros
- Si queremos cambiar a cadena de caracteres podemos hacerlo con la anotación `@Enumerated`

```
@Entity
@Table(name = "credit_card")
public class CreditCard {

    @Id
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    @Enumerated(EnumType.STRING)
    private CreditCardType creditCardType;

    // Constructors, getters, setters
}
```

```
public enum CreditCardType {
    VISA,
    MASTER_CARD,
    AMERICAN_EXPRESS
}
```

# Java Persistence API

## Mapeo: atributos

- Si no queremos que un atributo sea almacenado en la BDDD lo podemos marcar con `@Transient`

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    // Constructors, getters, setters
}
```

Obsérvese el uso de clases adaptadoras de tipos básicos:  
`Integer` en lugar de `int`, `Long` en lugar de `long`, etc. ¿Por qué?

# Java Persistence API

## Mapeo: acceso a atributos

- En los ejemplos mostrados hasta ahora hemos anotado las variables de instancia de las clase entidad
- Como consecuencia, el proveedor de persistencia consultará y modificará dichas variables de instancia
- A esto se le llama acceso al “campo” (**field access**)
- Si queremos que el proveedor de persistencia acceda a los atributos a través de **getters** y **setters** tenemos que usar el acceso a la “propiedad” (**property access**)
- Esto se consigue ubicando las anotaciones en los **getters** de las propiedades definidas por la clase entidad
- Se asume que todas las entidades son componentes Java (**beans**) que tienen **getters** y **setters** definidos de acuerdo a como exige el convenio para los **Beans**

# Java Persistence API

## Mapeo: acceso a atributos

- De acuerdo con el convenio para componentes Java:
  - El **getter** (método que consulta el valor) de una propiedad de nombre **X** de tipo **T** debe tener signatura:
    - **T getX()**
  - Opcionalmente, si **T=boolean** el nombre puede ser
    - **boolean isX()**
  - El **setter** (método que establece el valor) de una propiedad de nombre **X** y tipo **T** debe tener signatura:
    - **void setX(T valor)**
  - Si la propiedad es de solo lectura, no debe existir el **setter**
- En JPA si queremos utilizar acceso a las propiedades las anotaciones correspondientes a estas deben ir en el **getter** de dicha propiedad

# Java Persistence API

## Mapeo: acceso a atributos

- Ejemplo de acceso a campos

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    @Column(name = "first_name", nullable = false, length = 50)
    private String firstName;
    @Column(name = "last_name", nullable = false, length = 50)
    private String lastName;
    private String email;
    @Column(name = "phone_number", length = 15)
    private String phoneNumber;

    // Constructors, getters, setters
}
```

# Java Persistence API

## Mapeo: acceso a atributos

### Ejemplo de acceso a propiedades

```
@Entity
public class Customer {

    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;

    // Constructors

    @Id @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id= id;
    }

    @Column(name = "first_name", nullable = false, length = 50)
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column(name = "last_name", nullable = false, length = 50)
    public String getLastName() {
        return lastName;
    }
}
```

```
public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}

@Column(name = "phone_number", length = 15)
public String getPhoneNumber() {
    return phoneNumber;
}
public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
}
}
```

# Java Persistence API

## Mapeo: acceso a atributos

- Es posible indicar explícitamente qué tipo de acceso queremos para toda la clase ...

```
@Entity  
@Access(AccessType.FIELD)  
public class Customer {
```

- ... o para un atributo concreto

```
@Access(AccessType.PROPERTY)  
@Column(name = "phone_number", length = 555)  
public String getPhoneNumber() {  
    return phoneNumber;  
}
```

# Java Persistence API

## Mapeo: acceso a atributos

- Pero debemos tener cuidado:

!!! Si se combinan los dos tipos de accesos en la misma clase sin especificar explícitamente un tipo de acceso el resultado es impredecible!!!

# Java Persistence API

## Mapeo: colecciones como atributos

- Si queremos modelar un atributo cuyo valor es una colección de valores usamos una colección de Java (normalmente **List** o **Set**) y la anotación **@ElementCollection**

Establece el nombre de la tabla para los valores de la colección

BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true

```
@Entity  
public class Book {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    private String isbn;  
    private Integer nbOfPage;  
    private Boolean illustrations;  
    @ElementCollection(fetch = FetchType.LAZY)  
    @CollectionTable(name = "Tag")  
    @Column(name = "Value")  
    private List<String> tags = new ArrayList<>();  
  
    // Constructors, getters, setters  
}
```

TAG		
#BOOK_ID	bigint	Nullable = false
VALUE	varchar(255)	Nullable = true

# Java Persistence API

## Mapeo: colecciones como atributos

- También es posible usar un Map

Nombre de la columna que actúa como valor en el Map

```
@Entity  
public class CD {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    @Lob  
    private byte[] cover;  
    @ElementCollection  
    @CollectionTable(name="track")  
    @MapKeyColumn (name = "position")  
    @Column(name = "title")  
    private Map<Integer, String> tracks = new HashMap<>();  
  
    // Constructors, getters, setters  
}
```

Nombre de la columna que actúa como clave en el Map

CD		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
COVER	blob(64000)	Nullable = true

TRACK		
#CD_ID	bigint	Nullable = false
POSITION	integer	Nullable = true
TITLE	varchar(255)	Nullable = true

1O---O

# Java Persistence API

## Mapeo: embeddables

- Podemos definir una serie de atributos en una clase y luego incrustarlos en una entidad
- Ya lo hemos visto antes para claves compuestas

```
@Embeddable  
public class Address {  
  
    private String street1;  
    private String street2;  
    private String city;  
    private String state;  
    private String zipcode;  
    private String country;  
  
    // Constructors, getters, setters  
}
```



```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    @Embedded  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

```
create table CUSTOMER (  
    ID BIGINT not null,  
    LASTNAME VARCHAR(255),  
    PHONENUMBER VARCHAR(255),  
    EMAIL VARCHAR(255),  
    FIRSTNAME VARCHAR(255),  
    STREET2 VARCHAR(255),  
    STREET1 VARCHAR(255),  
    ZIPCODE VARCHAR(255),  
    STATE VARCHAR(255),  
    COUNTRY VARCHAR(255),  
    CITY VARCHAR(255),  
    primary key (ID)  
);
```

# Java Persistence API

## Mapeo: relaciones entre entidades

- En el dominio de las BBDD relacionales las relaciones entre entidades se modelan mediante el uso de claves foráneas

Customer				Address			
Primary key	Firstname	Lastname	Foreign key	Primary key	Street	City	Country
1	James	Rorisson	11				
2	Dominic	Johnson	12				
3	Maca	Macaron	13				



Repaso: ¿qué es una relación uno a muchos?

Customer		Address	
Primary key	Firstname	Primary key	Street
1	James	11	Aligre
2	Dominic	12	Balham
3	Maca	13	Alfama

Join table

Customer PK	Address PK
1	11
2	12
3	13

# Java Persistence API

## Mapeo: relaciones entre entidades

- En el mundo de la OO las asociaciones entre clases tienen cardinalidad y navegación



Repaso: ¿cómo se manifiesta la navegación?

Unidireccional



Bidireccional



# Java Persistence API

## Mapeo: relaciones entre entidades

- En total podemos encontrarnos ante 7 posibles combinaciones de **cardinalidad** y **navegación** para una asociación

<b>Cardinality</b>	<b>Direction</b>
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

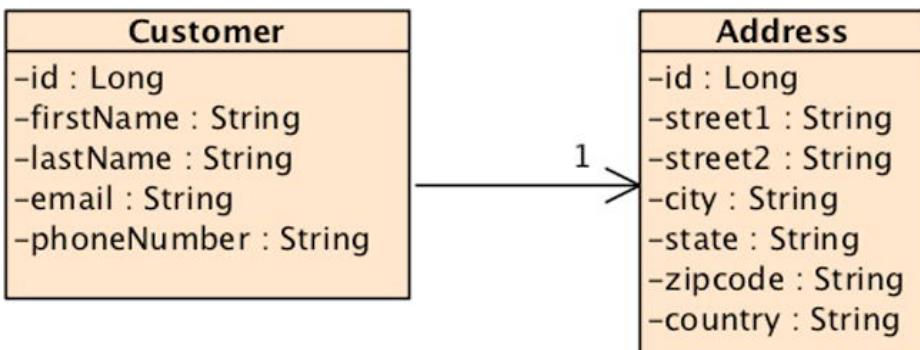


¿Por qué 7 y no 8?

# Java Persistence API

## Mapeo: relaciones entre entidades

- Si una entidad A está relacionada con UNA (a lo sumo) entidad B, la clase de entidad A tendrá una referencia a la clase de entidad B



```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

# Java Persistence API

## Mapeo: relaciones entre entidades

- Si una entidad A está relacionada con **MUCHAS** entidades B, la clase de entidad A tendrá una colección de referencias a la clase de entidad B



```
@Entity  
public class Order {  
  
    @Id @GeneratedValue  
    private Long id;  
    @Temporal(TemporalType.TIMESTAMP)  
    private Date creationDate;  
    private List<OrderLine> orderLines;  
  
    // Constructors, getters, setters  
}
```

# Java Persistence API

## Mapeo: relaciones entre entidades

- Para cada relación entre entidades, el proveedor de persistencia modelará la relación mediante las tablas y columnas necesarias para poder guardar información acerca de la relación

```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

```
create table CUSTOMER (  
    ID BIGINT not null,  
    FIRSTNAME VARCHAR(255),  
    LASTNAME VARCHAR(255),  
    EMAIL VARCHAR(255),  
    PHONENUMBER VARCHAR(255),  
    ADDRESS_ID BIGINT,  
    primary key (ID),  
    foreign key (ADDRESS_ID) references ADDRESS(ID)  
)
```

# Java Persistence API

## Mapeo: relaciones entre entidades

- Si queremos cambiar algún parámetro del mapeo por defecto (el nombre de la clave foránea, por ejemplo) podemos usar anotaciones

@OneToOne indica que el atributo representa una relación uno-a-uno

También existen @OneToMany, @ManyToOne y @ManyToMany

La clave foránea se llamará add\_fk

```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    @OneToOne (fetch = FetchType.LAZY)  
    @JoinColumn(name = "add_fk", nullable = false)  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

La dirección se consulta en la BBDD cuando se acceda a ella

# Java Persistence API

## Mapeo: relaciones entre entidades

- En las relaciones **bidireccionales** tenemos que indicar mediante anotaciones qué atributos de sendas entidades forman parte de la misma relación
- Si no lo hacemos el proveedor de persistencia pensará que son dos relaciones unidireccionales

```
@Entity  
public class Artist {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    @ManyToMany  
  
    private List<CD> appearsOnCDs ;  
  
    // Constructors, getters, setters  
}
```

Propietaria de la relación

```
@Entity  
public class CD {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    @ManyToMany(mappedBy = "appearsOnCDs")  
    private List<Artist> createdByArtists;  
  
    // Constructors, getters, setters  
}
```

La entidad propietaria establece los parámetros de la relación (nombres de columnas, tablas, etc)

En una relación unidireccional solo hay una propietaria  
En una relación bidireccional uno-a-uno o muchos-a-muchos debemos elegir nosotros  
En una relación bidireccional uno-a-muchos la propietaria es la que está en el “muchos”

# Java Persistence API

## Mapeo: relaciones entre entidades

- ¿Cómo se modelaría una relación muchos-a-muchos?

```
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", -
               joinColumns = @JoinColumn(name = "artist_fk"), -
               inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

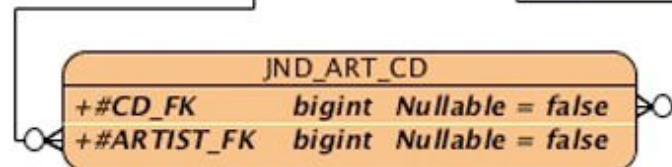
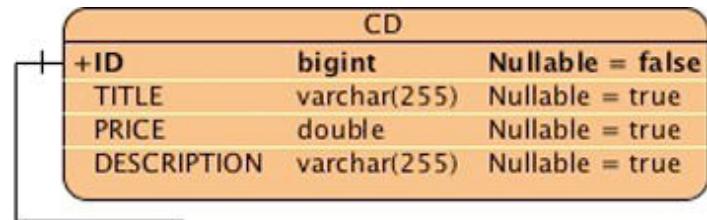
    // Constructors, getters, setters
}
```



```
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```



# Java Persistence API

## Mapeo: relaciones entre entidades

- Ejemplo de relación uno-a-muchos bidireccional

```
@Entity  
public class Grupo implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    private String nombre;  
    @OneToMany (mappedBy="grupo")  
    private List<Alumno> alumnos;
```



¿Quién es propietaria  
de la relación?

```
@Entity  
public class Alumno implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    private String nombre;  
    private String apellidos;  
    private String dni;  
    private String expediente;  
    @ManyToOne  
    private Grupo grupo;
```



¿Qué esquema se  
espera en la BBDD?

# Java Persistence API

## Mapeo: relaciones entre entidades

- Por defecto cada combinación **cardinalidad/navegación** se modela de la siguiente forma:

<b>Cardinality</b>	<b>Direction</b>
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Clave foránea en propietaria  
Clave foránea en propietaria  
Tabla de unión  
Clave foránea en propietaria  
Clave foránea en propietaria  
Tabla de unión  
Tabla de unión



Adivina, adivinanza

# Java Persistence API

## Mapeo: relaciones entre entidades

- Ejercicio
  - Una empresa cárnica nos pide desarrollar una aplicación de trazabilidad alimentaria para guardar la traza de todos los productos que elaboran. La empresa elabora muchos tipos de productos. Cada lote de productos tiene asociada una referencia. Por razones sanitarias la empresa debe guardar para cada lote la fecha de elaboración, la fecha de consumo preferente, las referencias a los lotes de los ingredientes empleados y los clientes a quienes ha vendido cantidades superiores a 5Kg del producto
  - Modelar la relación entre lote y producto usando entidades JPA

# Java Persistence API

## Mapeo: herencia

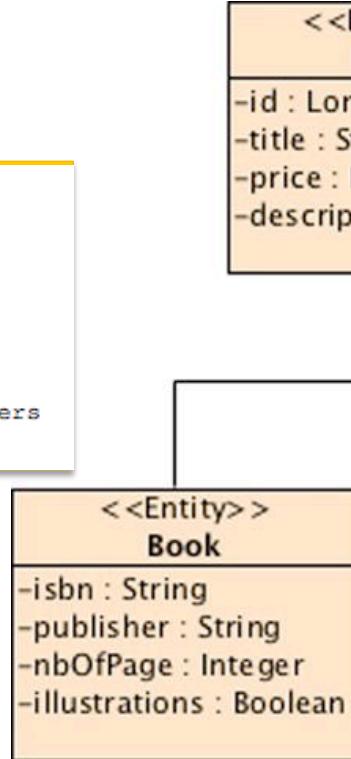
- La herencia es un concepto que existe en OO pero no en BBDD relacionales
- JPA ofrece tres estrategias para modelar la herencia:
  - Una tabla por jerarquía de entidades (*single-table-per-class hierarchy*): `InheritanceType.SINGLE_TABLE`
  - Una tabla por clase de la jerarquía con vínculos entre ellas (*joined-subclass*): `InheritanceType.JOINED`
  - Una tabla por clase concreta de la jerarquía (*table-per-concrete-class*): `InheritanceType.TABLE_PER_CLASS`

# Java Persistence API

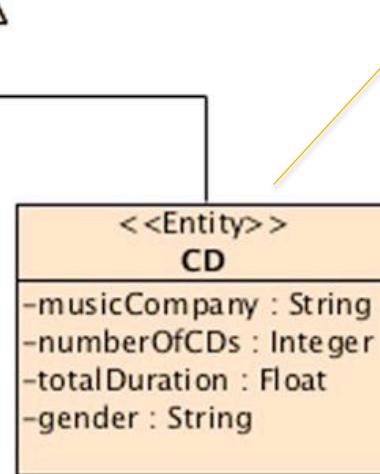
## Mapeo: herencia

### Ejemplo

```
@Entity  
public class Book extends Item {  
  
    private String isbn;  
    private String publisher;  
    private Integer nbOfPage;  
    private Boolean illustrations;  
  
    // Constructors, getters, setters  
}
```



```
@Entity  
public class Item {  
  
    @Id @GeneratedValue  
    protected Long id;  
    protected String title;  
    protected Float price;  
    protected String description;  
  
    // Constructors, getters, setters  
}
```



```
@Entity  
public class CD extends Item {  
  
    private String musicCompany;  
    private Integer numberOfCDs;  
    private Float totalDuration;  
    private String genre;  
  
    // Constructors, getters, setters  
}
```

# Java Persistence API

## Mapeo: herencia

- Estrategia de tabla única
  - Es la estrategia por defecto

Esta columna determina  
cuál es la clase de la  
entidad (discriminador)



ID	DTYPE	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	Item	Pen	2.10	Beautiful black pen			...
2	CD	Soul Trane	23.50	Fantastic jazz album	Prestige		...
3	CD	Zoot Allures	18	One of the best of Zappa	Warner		...
4	Book	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	Book	H2G2	17.50	Funny IT book ;o)		1-278-983	...

ITEM		
+ID	bigint	Nullable = false
DTYPE	varchar(31)	Nullable = true
TITLE	varchar(255)	Nullable = false
PRICE	double	Nullable = false
DESCRIPTION	varchar(255)	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENRE	varchar(255)	Nullable = true

# Java Persistence API

## Mapeo: herencia

- Estrategia de tabla única
  - Podemos cambiar el tipo y la columna del discriminador

```
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn (name="disc", -  
                      discriminatorType = DiscriminatorType.CHAR)  
@DiscriminatorValue("I")  
public class Item {
```

CHAR	DiscriminatorType
INTEGER	DiscriminatorType
STRING	DiscriminatorType

```
@Entity  
@DiscriminatorValue("B")  
public class Book extends Item {
```

```
@Entity  
@DiscriminatorValue("C")  
public class CD extends Item {
```

ID	DISC	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	I	Pen	2.10	Beautiful black pen			...
2	C	Soul Trane	23.50	Fantastic jazz album	Prestige		...
3	C	Zoot Allures	18	One of the best of Zappa	Warner		...
4	B	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	B	H2G2	17.50	Funny IT book ;o)		1-278-983	...

# Java Persistence API

## Mapeo: herencia

- Estrategia de tablas vinculadas
  - Se indica `InheritanceType.JOINED` en la raíz de la jerarquía

```
@Entity  
@Inheritance(strategy = InheritanceType.JOINED )  
public class Item {
```

- Sigue existiendo una columna discriminadora
- Cada tabla contiene solo los atributos que son específicos de dicha entidad (no están en un ancestro)



# Java Persistence API

## Mapeo: herencia

### Estrategia de tablas por clase concreta

```
@Entity  
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS )  
public class Item {
```

- Se crea una tabla por cada clase concreta (no abstracta)
- Ya no hay discriminador
- No hay vínculos entre tablas
- No puede haber dos valores iguales de la clave primaria en tablas distintas
- Esta estrategia es opcional y **no tiene por qué estar soportada por el proveedor de persistencia**

BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true

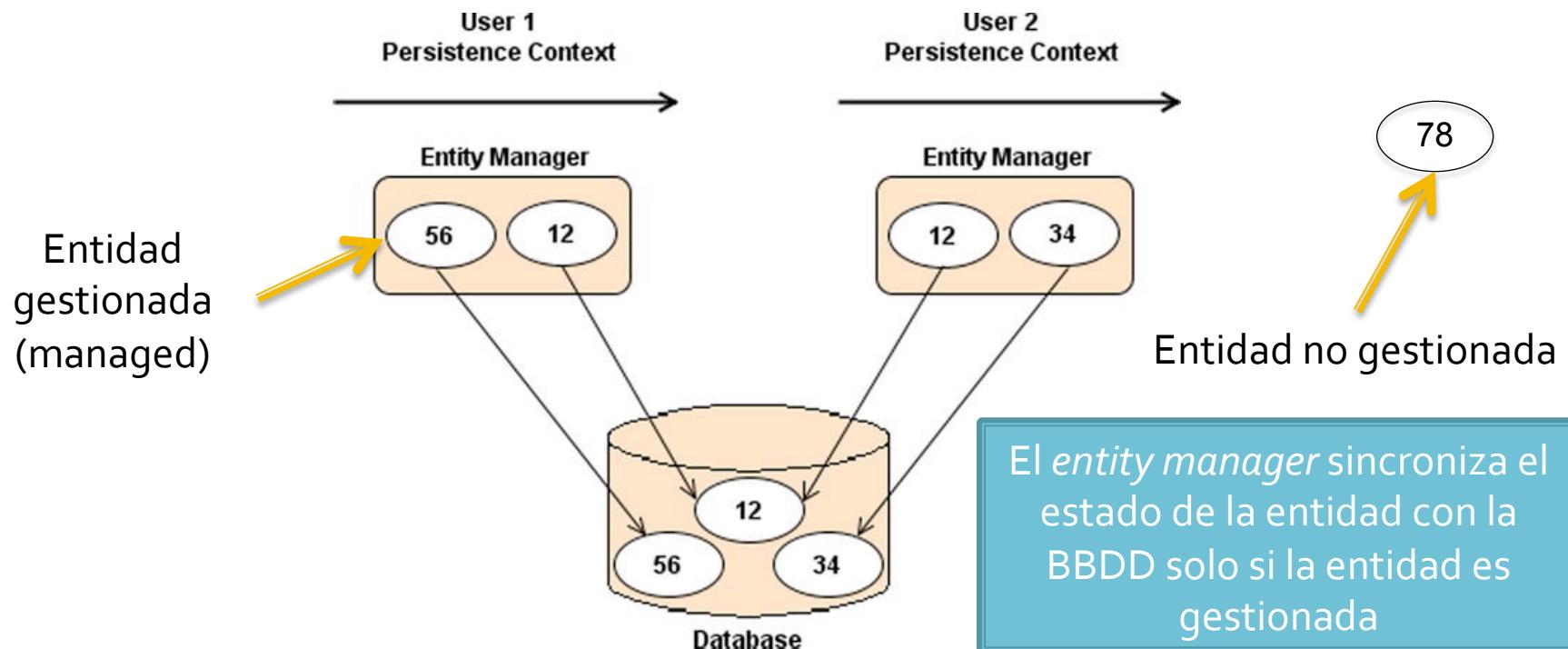
ITEM		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true

CD		
+ID	bigint	Nullable = false
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TITLE	varchar(255)	Nullable = true
TOTALDURATION	double	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
GENRE	varchar(255)	Nullable = true

# Java Persistence API

## Operaciones: contexto de persistencia

- Un contexto de persistencia (*persistent context*) es un conjunto de entidades (donde solo hay una entidad por cada identidad) gestionado por un gestor de entidades (*entity manager*)



# Java Persistence API

## Operaciones: contexto de persistencia

- Las entidades se crean como cualquier otro objeto Java (puesto que son POJOs)
- Para introducirlas (o eliminarlas) en un contexto de persistencia usamos los métodos de la interfaz `EntityManager`
- La forma de obtener una implementación de esta interfaz depende del contexto en que se ejecuta la aplicación
  - En un entorno gestionado por la aplicación
    - Obtenemos un objeto que implementa `EntityManagerFactory`
    - Obtenemos un `EntityManager` a partir de él
  - En un entorno gestionado por el contenedor (e.g., en Glassfish)
    - Anotamos una variable `EntityManager` con `@PersistentContext` y el contenedor inyectará la referencia adecuada

# Java Persistence API

## Operaciones: contexto de persistencia

### ■ Entorno gestionado por la aplicación

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPAppPU");  
EntityManager em = emf.createEntityManager();
```

- En este entorno debemos marcar las transacciones explícitamente

```
EntityTransaction tx = em.getTransaction();  
tx.begin();  
  
// Operaciones con entidades  
  
tx.commit();
```

### ■ Entorno gestionado por el contenedor

```
@PersistenceContext(unitName="JPAppPU")  
private EntityManager em;
```

En este entorno no necesitamos marcar las transacciones

# Java Persistence API

## Operaciones: contexto de persistencia

- El nombre de la unidad de persistencia se establece en el fichero de configuración persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="JPAAppPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>jpaapp.modelo.Traza</class>
        <class>jpaapp.modelo.Producto</class>
        <class>jpaapp.modelo.Ingrediente</class>
        <class>jpaapp.modelo.Address</class>
        <class>jpaapp.modelo.Coordenada</class>
        <class>jpaapp.modelo.Grupo</class>
        <class>jpaapp.modelo.Alumno</class>
        <class>jpaapp.modelo.Autor</class>
        <class>jpaapp.modelo.Articulo</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/sample"/>
            <property name="javax.persistence.jdbc.password" value="app"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.user" value="app"/>
            <property name="javax.persistence.schema-generation.scripts.action" value="create"/>
            <property name="javax.persistence.schema-generation.scripts.create-target" value="/tmp/create-traza.ddl"/>
        </properties>
    </persistence-unit>
</persistence>
```

Transacciones gestionadas por la aplicación  
(JTA cuando sea gestionado por el contenedor)

Clases de entidad

Parámetros de conexión a la BBDD, generación de scripts, etc.

# Java Persistence API

## Operaciones: inserción

- Para insertar una entidad en la BBDD usamos el método **persist** (de EntityManager)

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

- Si la clave es autogenerada tomará valor tras el **commit()**
- Los objetos pasan a formar parte del contexto de persistencia (si no lo eran)

# Java Persistence API

## Operaciones: eliminación

- Para eliminar una entidad de la BBDD usamos el método `remove` (de `EntityManager`)

```
tx.begin();
em. remove (customer);
tx.commit();
```

- Si la entidad tiene una relación con otra entidad debemos decidir si la operación (en este caso eliminación) se propaga a la otra entidad

```
@OneToOne (fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST, CascadeType.REMOVE} )
@JoinColumn(name = "address_fk")
private Address address;
```

- Si alguna entidad queda “huérfana” podemos configurar la relación para que sea eliminada

```
@OneToOne (fetch = FetchType.LAZY, orphanRemoval=true )
private Address address;
```

# Java Persistence API

## Operaciones: actualización

- Basta con establecer el nuevo valor para el atributo dentro de una transacción

```
tx.begin();
customer.setFirstName("Williman");

tx.commit();
```

- Para actualizar una relación basta con modificar las listas (o referencias) de las entidades
- El proveedor de persistencia consultará la entidad que sea **propietaria** de la relación

# Java Persistence API

## Operaciones: actualizar de la BBDD

- Si queremos actualizar una entidad con los valores que aparecen en la BBDD (para deshacer un cambio, por ejemplo) usamos `refresh()`

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), " Antony ");

customer.setFirstName(" William ");

em.refresh(customer);
assertEquals(customer.getFirstName(), " Antony ");
```

# Java Persistence API

## Operaciones: consultar por clave primaria

- Si conocemos la clave primaria de una entidad podemos usar el método `find()` para obtener dicha entidad

```
Customer customer = em.find (Customer.class, 1234L)
if (customer!= null) {
    // Process the object
}
```

- Una vez que tenemos la entidad podemos consultar sus atributos y entidades relacionadas con los `getters`

```
Grupo g = em.find(Grupo.class, 1L);
System.out.println("Alumnos del grupo "+g.getNombre());
System.out.println("-----");

for (Alumno a: g.getAlumnos())
{
    System.out.println(a.getNombre());
}
```

# Java Persistence API

## Operaciones: tipos de carga de datos

- Cuando el EM consulta entidades en la BBDD no siempre carga toda la información
- Por eficiencia, a veces devuelve un objeto “referencia” que consulta la BBDD automáticamente cuando es accedido
- Ejemplo

```
Grupo g = em.find(Grupo.class, 1L);
System.out.println("Alumnos del grupo "+g.getNombre());
System.out.println("-----");
for (Alumno a: g.getAlumnos())
{
    System.out.println(a.getNombre());
}
```

Mientras no se acceda a la lista no se cargarán los datos de los alumnos de un grupo (se evitan cargas innecesarias y costosas)

# Java Persistence API

## Operaciones: tipos de carga de datos

- A esto se le llama *carga perezosa* (*lazy fetch*)
- Por defecto los tipos de carga de las relaciones son

<b>Annotation</b>	<b>Default Fetching Strategy</b>
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

- Pero podemos modificarla dando un valor al atributo *fetch* de la anotación

```
@OneToMany(fetch = FetchType.EAGER )  
private List<OrderLine> orderLines;
```

# Java Persistence API

## Operaciones: manipular el contexto

- Hasta ahora hemos visto cómo insertar un POJO en el contexto de persistencia para que sea gestionado
- Podemos consultar si una entidad es gestionada por un contexto de persistencia con el método `contains()`

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

tx.begin();
em.remove(customer);
tx.commit();

assertFalse(em.contains(customer));
```

# Java Persistence API

## Operaciones: manipular el contexto

- Podemos transformar una entidad gestionada en un simple POJO
- El método `detach()` hace eso

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

assertTrue (em.contains(customer));

em.detach(customer);

assertFalse (em.contains(customer));
```

- El método `clear()` limpia el contexto de persistencia completo (lo deja vacío)

# Java Persistence API

## Operaciones: mezclando entidades

- En ocasiones tenemos una entidad no gestionada cuya información nos gustaría transferir a una entidad gestionada con el mismo ID
- Esto ocurre en particular, cuando la entidad no gestionada proviene de otro contenedor en un servidor de aplicaciones (e.g., de una página JSF)
- El método `merge()` permite “unir” la información de la entidad no gestionada con una gestionada y devuelve la entidad gestionada

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

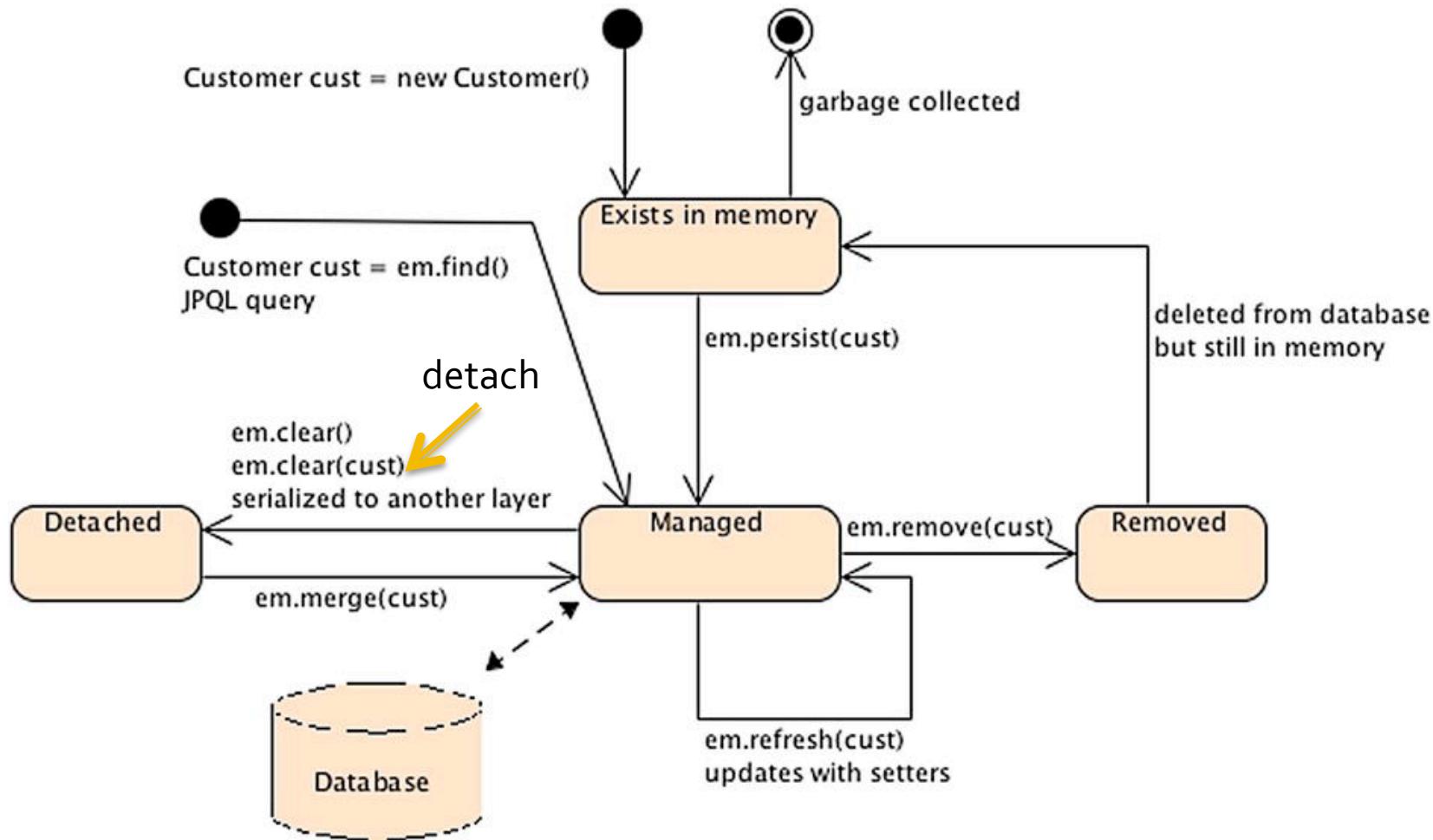
em.clear();  Limpiamos el contexto de persistencia para simular el fin de una transacción en un contexto gestionado por el contenedor (e.g., EJB)

// Sets a new value to a detached entity
customer.setFirstName("William");

tx.begin();
em.merge(customer);
tx.commit();
```

# Java Persistence API

## Operaciones: ciclo de vida de las entidades



# Java Persistence API

## Operaciones: cascada

- Si queremos que una operación aplicada a una entidad se propague a las entidades con las que está relacionada podemos establecer el atributo **cascade** de la anotación asociada a la relación

```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    @OneToOne (fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST, CascadeType.REMOVE})  
    @JoinColumn(name = "address_fk")  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

Cuando una entidad `Customer` sea insertada o eliminada la dirección asociada con él también lo será (en otro caso tendríamos que hacer la inserción o eliminación nosotros)

Cascade Type	Description
PERSIST	Cascades persist operations to the target of the association
REMOVE	Cascades remove operations to the target of the association
MERGE	Cascades merge operations to the target of the association
REFRESH	Cascades refresh operations to the target of the association
DETACH	Cascades detach operations to the target of the association
ALL	Declares that all the previous operations should be cascaded

# Java Persistence API

## Operaciones: JPQL

- *Java Persistence Query Language* (JPQL) es un lenguaje de consulta similar a SQL pero basado en entidades en lugar de tablas

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

- Ejemplos:

```
SELECT c
FROM Customer c
```

```
SELECT c.firstName, c.lastName
FROM Customer c
```

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Vincent' AND c.address.country = 'France'
```

```
SELECT NEW org.agoncal.javaee7.CustomerDTO(c.firstName, c.lastName, c.address.street1)
FROM Customer c
```

# Java Persistence API

## Operaciones: JPQL

- ¿Cómo lo usamos?
  - Consultas dinámicas (el más simple y directo)

```
Query query = em.createQuery ("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList ();

TypedQuery <Customer> query = em.createQuery("SELECT c FROM Customer c", Customer.class );
List<Customer> customers = query.getResultList();
```

- Pueden tener parámetros
  - Con nombre

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = :fname ");
query.setParameter(" fname ", "Betty");
List<Customer> customers = query.getResultList();
```

- Sin nombre

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1 ");
query.setParameter( 1 , "Betty");
List<Customer> customers = query.getResultList();
```

# Java Persistence API

## Operaciones: JPQL

- ¿Cómo lo usamos?
  - Consultas con nombre

```
@Entity
@NamedQueries({
    @NamedQuery ( name = " findAll ", query ="select c from Customer c"),
    @NamedQuery(name = "findVincent", →
                query="select c from Customer c where c.firstName = 'Vincent'"),
    @NamedQuery(name = "findWithParam", →
                query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
```

- Son estáticas pero pueden ser más eficientes
- Se definen con anotaciones asociadas a una entidad

# Java Persistence API

## Operaciones: JPQL

- ¿Cómo lo usamos?
  - Consultas con nombre (cont.)
    - Las invocamos así

Nombre de la consulta



```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");
query. setMaxResults (3);
List<Customer> customers = query.getResultList();
```

  - *Query* tiene una interfaz *fluent* que hace el código más legible

```
Query query = em.createNamedQuery("findWithParam").setParameter("fname", "Vincent")
                .setMaxResults(3);
```

# Java Persistence API

## Operaciones: bloqueos

- Problema: ¿qué pasa si tenemos transacciones concurrentes modificando la misma entidad?

```
tx1.begin();
// The price of the book is 10$  
Book book = em.find(Book.class, 12);
book.raisePriceByTwoDollars();
tx1.commit();
// The price is now 12$
```

```
tx2.begin();
// The price of the book is 10$  
Book book = em.find(Book.class, 12);
book.raisePriceByFiveDollars();
tx2.commit();
// The price is now 15$
```

time  
↓

Quien acabe después “gana”

# Java Persistence API

## Operaciones: bloqueos

- Para evitar esto JPA permite utilizar dos tipos de bloqueos:
  - Pesimista: se basa en los mecanismos de bloqueo de la BBDD subyacente
  - Optimista: se basa en un método de versiones (recomendado para poca concurrencia)
- Para utilizar un bloqueo optimista basta con añadir un atributo “versión” anotado con `@Version` a las entidades

# Java Persistence API

## Operaciones: bloqueos

- Cada vez que se actualiza una fila aumenta su versión

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    @Version
    private Integer version;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}

tx.begin();
em.persist(book);
tx.commit();
assertEquals( 1, book.getVersion() );

tx.begin();
book.raisePriceByTwoDollars();
tx.commit();
assertEquals( 2, book.getVersion() );
```

# Java Persistence API

## Operaciones: bloqueos

- Si al actualizar se descubre que la versión de la entidad en la BBDD y en el EM no coinciden, se produce una excepción y se deshace la transacción (rollback)

```
tx1.begin();

// The price of the book is 10$
Book book = em.find(Book.class, 12);
// book.getVersion() == 1

book.raisePriceByTwoDollars();

tx1.commit();
// The price is now 12$
// book.getVersion() == 2
```

time ↓

```
tx2.begin();

// The price of the book is 10$
Book book = em.find(Book.class, 12);
// book.getVersion() == 1

book.raisePriceByFiveDollars();

tx2.commit();
// version should be 1 but is 2
// OptimisticLockException
```

# Para ampliar conocimientos

- Antonio Goncalves, Beginning Java EE 7 (cap. 4, 5 y 6)
- Especificación de JPA 2.1 (JSR 338):  
<https://jcp.org/en/jsr/detail?id=338>
- Oracle Java EE 7 Tutorial (parte VIII, caps.37-44)