An abstract graphic on the left side of the slide. It features a close-up, angled view of a computer keyboard. Several keys are highlighted with glowing arrows pointing to the right. The arrows are in various colors: white, light blue, green, and orange. The background of the keyboard is dark, and the overall lighting is warm, with a gradient from dark blue at the top to bright orange at the bottom.


# Herramientas de mapeado objeto-relacional (ORM)

Módulo: Acceso a datos

Ciclo: Desarrollo de Aplicaciones  
Multiplataforma

- ♦ Del 9 de enero al 24 ☐ Realización de ejercicios resueltos
  - ♦ Del 27 de enero al 16 de febrero ☐ Realización del proyecto
  - ♦ Entrega del proyecto ☐ Día 16 de febrero
  - ♦ Día 13 de febrero ☐ Examen UD 3
-

# Herramientas ORM

- ♦ Propósito ORM: convertir datos entre lenguaje POO y lenguaje SQL
  - ♦ Herramientas ORM:
    - ♦ Mapean tablas a clases y columnas a atributos
    - ♦ Automatizan operaciones CRUD
    - ♦ Gestionan relaciones
    - ♦ Consultas avanzadas
    - ♦ Manejo de sesiones y transacciones
  - ♦ Se conectan a BD relacionales, extraen la información hacia objetos.
  - ♦ Para ello, se necesita:
    - ♦ Definir la correspondencia entre clase y entidad.
    - ♦ Utilizar **POJOs** 
    - ♦ Indicar a la ORM que los haga persistentes.
- POJOs:
- Plain Old Java Objects
  - Clases simples que no dependen de un framework
  - Clases Java que no extienden ni implementan nada.

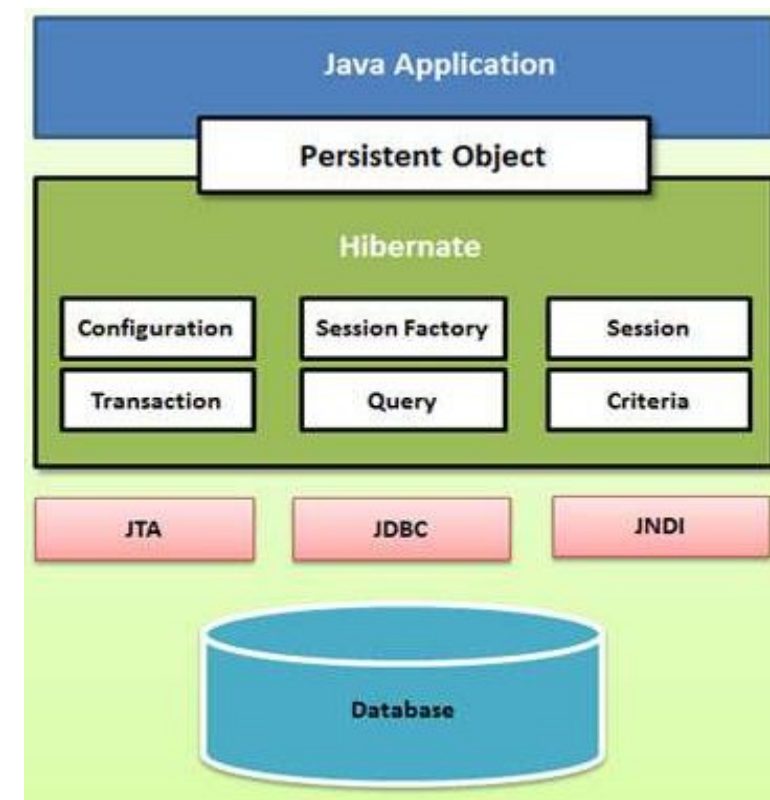
# Herramientas ORM

- ♦ Ventajas:
  - ♦ Reducen tiempo de desarrollo
  - ♦ Permite la abstracción e independencia de la BD
  - ♦ Facilitan la reutilización
  - ♦ Incrementa la portabilidad y escalabilidad
- ♦ Desventajas:
  - ♦ Requieren tiempo para su aprendizaje □ son complejos.
  - ♦ Ofrecen menor rendimiento □ aplicaciones más lentas
    - ♦ Leer y transformar las consultas de POO a BD
    - ♦ Leer registros
    - ♦ Crear objetos
- ♦ Ejemplos:
  - ♦ **Hibernate**
  - ♦ JPA
  - ♦ iBatis

# Arquitectura Hibernate

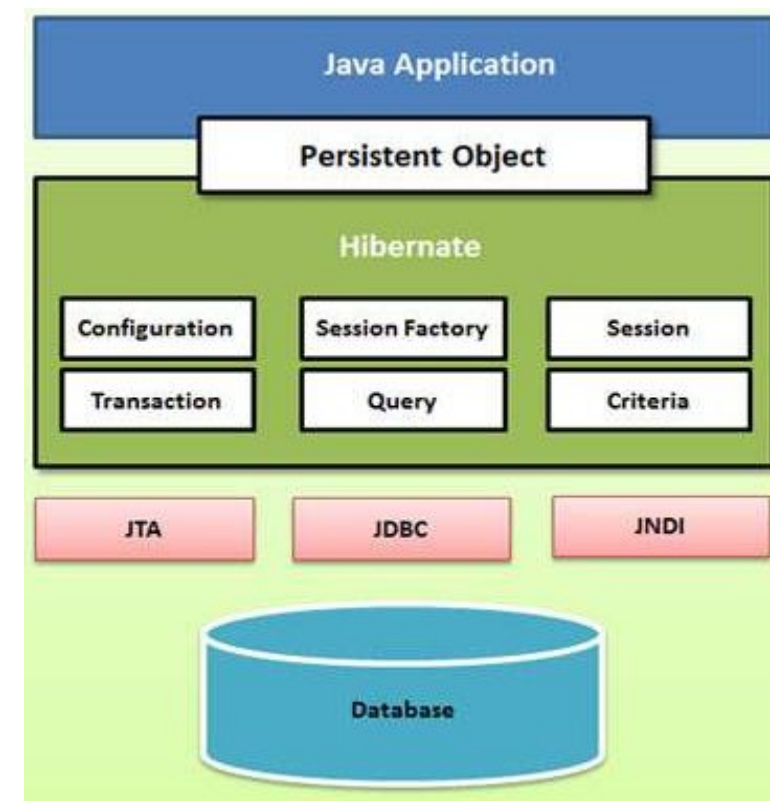
5

- ♦ Arquitectura en capas:
- ♦ **Configuration**
  - ♦ Representa el archivo de configuración o propiedades
  - ♦ Proporciona dos componentes claves:
    - ♦ Conexión de bases de datos
    - ♦ Configuración del mapeado de clase.
- ♦ **Session Factory:**
  - ♦ Configura Hibernate para utilizar el archivo de configuración
  - ♦ Crea instancia objeto Session.
- ♦ **Session:**
  - ♦ Permite obtener una conexión física con una base de datos.



# Arquitectura Hibernate

- ♦ Arquitectura en capas:
- ♦ **Transaction:**
  - ♦ Representa una unidad de trabajo con la BD.  $\Rightarrow$  Transacción
  - ♦ Transacciones manejadas por un gestor de transacciones
    - ♦ Ejemplo: **JDBC** o JTA.
- ♦ **Query:**
  - ♦ Objetos de consulta
  - ♦ Cadena Hibernate Query Language (HQL)
  - ♦ Cadena SQL



# Instalación

- ◆ Instalación con Maven ⇒ pom.xml

- ◆ **groupId**: identifica el proyecto de forma única entre proyectos

- ◆ **artifactId**: nombre que tomaría el JAR

- ◆ **dependencies**:

- ◆ sección más importante

- ◆ añadimos las librerías

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>Tema3</groupId>
  <artifactId>Tema3</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <!-- Dependencias-->
  </dependencies>
</project>
```

- ♦ Instalación de Hibernate con Maven ⇒ pom.xml

- ♦ Hibernate-core

- ♦ Mysql

- ♦ **Actualizar dependencias**

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>6.6.4.Final</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.22</version>
</dependency>
```




# Configuración

- ◆ Archivo hibernate.cfg.xml

- ◆ En src/

```
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/empleados?createDatabaseIfNotExist=true&serverTimezone=UTC</property>
    <property name="connection.username">root</property>
    <property name="connection.password">abc123.</property>
    <property name="hbm2ddl.auto">create</property>
    <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
    <property name="hibernate.dialect.storage_engine">innodb</property>
    <property name="hibernate.show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```



Indica la forma de validación o exportación del esquema DDL cuando el objeto SessionFactory es creado:

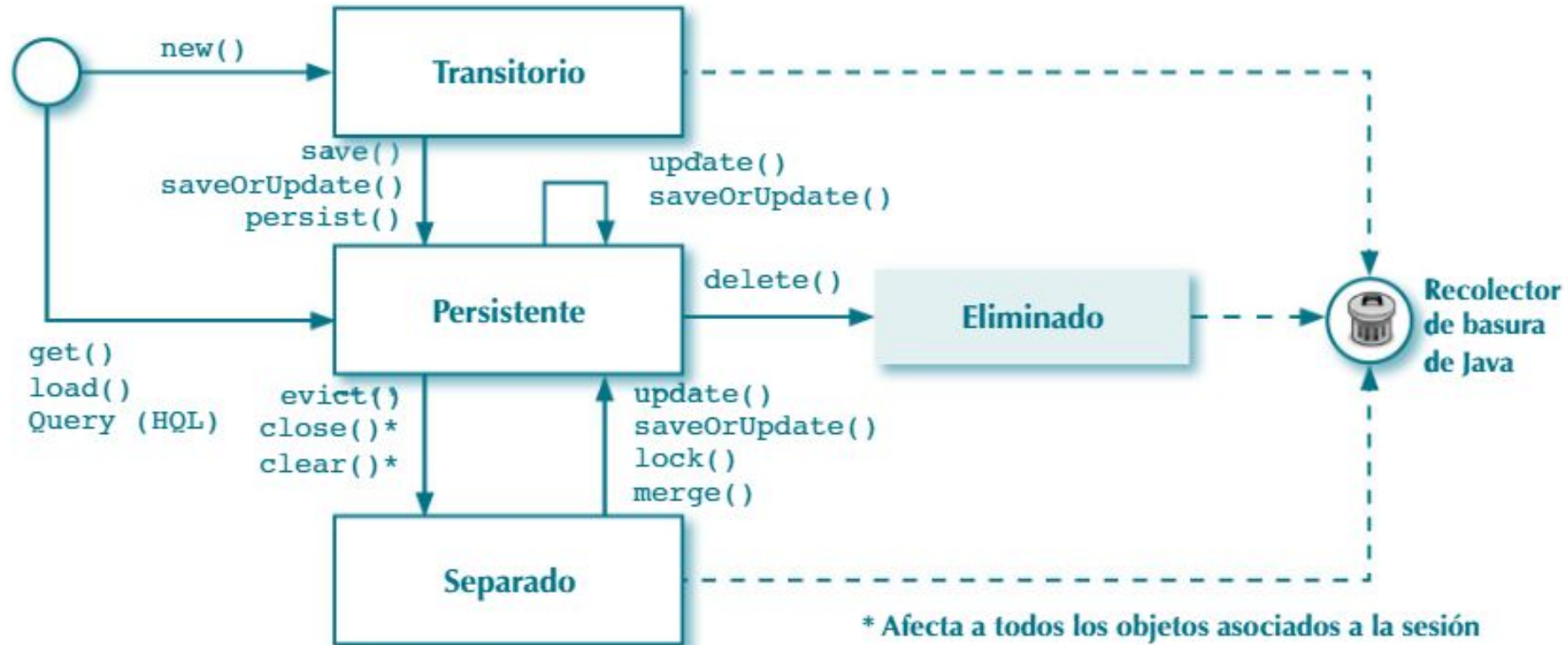
- Validate: valida esquema, no hace cambios
- Create-only: genera BD
- Drop: elimina la BD
- **Update**: actualiza el esquema
- Create: crea esquema, destruye anterior
- Create-drop: elimina esquema al cerrar el objeto
- None: no hace nada

# Sesiones y estados de un objeto

- ♦ **Session:** un objeto, representa una unidad de trabajo con la BD
- ♦ Permite representar el gestor de persistencia.
- ♦ Formado internamente por una cola de sentencias SQL para sincronizar el estado.
- ♦ Estados:
  - ♦ **Transitorio:** recién creado, no enlazado con el gestor de persistente
  - ♦ **Persistente:** objeto enlazado con la sesión y los cambios será persistentes
  - ♦ **Disociado:** está en memoria y existe tanto en Java como en la BD
  - ♦ **Borrado:** objeto marcado para ser borrado, existe en Java y se borrará de la BD.

# Sesiones y estados de un objeto

11



# Prueba básica con Hibernate

---

- ♦ Creación fichero HibernateUtil.java ☐ Patrón de diseño?
- ♦ Creación fichero main.java

# Mapeo de objetos

- ♦ Definiciones:
  - ♦ **Persistencia de objetos:** capacidad de guardar y recuperar un objeto de un medio de almacenamiento.
  - ♦ **Clases persistentes:** clases que permiten representar entidades de la BD.
- ♦ Dos formas de realizar el mapeo:
  - ♦ Ficheros *nombre\_entidad.hbm.xml*
  - ♦ Utilizando anotaciones en Java

# Ficheros *nombre\_entidad.hbm.xml*

14

```
<hibernate-mapping>
<class name="Persistente.Departamentos" table="DEPARTAMENTOS" schema="ANTONIO">
  <id name="deptNo" type="byte">
    <column name="DEPT_NO" precision="2" scale="0" />
    <generator class="assigned" />
  </id>
  <property name="dnombre" type="string">
    <column name="DNOMBRE" length="15" />
  </property>
  <property name="loc" type="string">
    <column name="LOC" length="15" />
  </property>
  <set name="empleadoses" inverse="true">
    <key>
      <column name="DEPT_NO" precision="2" scale="0" not-null="true" />
    </key>
    <one-to-many class="Persistente.Empleados" />
  </set>
</class>
</hibernate-mapping>
```

```
public class Departamentos implements java.io.Serializable {

    private byte deptNo;

    private String dnombre;
    private String loc;
    private Set empleadoses = new HashSet(0);

    public Departamentos() {
    }

    public Departamentos(byte deptNo) {
        this.deptNo = deptNo;
    }

    public Departamentos(byte deptNo, String dnombre, String loc, Set empleadoses) {
        this.deptNo = deptNo;
        this.dnombre = dnombre;
        this.loc = loc;
        this.empleadoses = empleadoses;
    }

    public byte getDeptNo() {
        return this.deptNo;
    }

    public void setDeptNo(byte deptNo) {
        this.deptNo = deptNo;
    }

    public String getDnombre() {
        return this.dnombre;
    }

    public void setDnombre(String dnombre) {
        this.dnombre = dnombre;
    }

    public String getLoc() {
        return this.loc;
    }

    public void setLoc(String loc) {
        this.loc = loc;
    }

    public Set getEmpleadoses() {
        return this.empleadoses;
    }

    public void setEmpleadoses(Set empleadoses) {
        this.empleadoses = empleadoses;
    }

    public void setdeptNo(byte b) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

}
```

# Utilización de anotaciones

---

- ♦ **Anotación:** añaden metadatos en el código Java que están disponibles en tiempo de ejecución o de compilación.
  - ♦ Características:
    - ♦ Comienzan con @
    - ♦ No cambian la actividad del programa
    - ♦ Ayudan a relacionar metadatos con componentes
  - ♦ Hibernate establece que toda clase vinculada con anotaciones debe ser **Bean**
    - ♦ Constructor vacío obligatorio (puede tener más)
    - ♦ Disponer de propiedades privadas
    - ♦ Implementar Getters y Setters
-

- ♦ Librería utilizada para el desarrollo en Java
- ♦ **Objetivo:** evitar la escritura de código técnico repetitivo → Evitar boilerplate
- ♦ Hace uso de diferentes anotaciones:
  - ♦ `@Data`: getters, setters, toString, hashCode, equals
  - ♦ `@AllArgsConstructor`: constructor con todos los parámetros
  - ♦ `@NoArgsConstructor`: constructor vacío
  - ♦ `@RequiredArgsConstructor`: constructor con los parámetros que se hayan marcado con `@NotNull`
  - ♦ `@Builder`: otro tipo de constructor diferente al típico new
  - ♦ `@SuperBuilder`: para generación de constructores con herencia



# Ejemplos Lombok

- ◆ Ejemplo básico
- ◆ Ejemplo herencia
- ◆ Ejemplo builder

## ♦ Anotaciones Hibernate:

- ♦ **@Entity** (`name="usuarios"`): marca que la clase es una entidad
- ♦ **@Table** (`name = "usuarios"`): mapea clase Java con la tabla usuarios de la BD. Si no se establece el atributo name coge el nombre de la clase

## NOTA:

- ♦ Tanto @Entity como @Table disponen del atributo "name".
- ♦ En @Table, *name* permite identificar la tabla de la BD contra la que se va a mapear la Entidad
- ♦ En @Entity, *name* indica cómo se denominará la Entidad en las consultas

# Utilización de anotaciones

---

- ◆ **Anotaciones Hibernate:**

- ◆ **@Id**: permite identificar qué atributo de la clase será la clave primaria
- ◆ **@GeneratedValue (strategy = GenerationType.AUTO)**: permite especificar la estrategia por defecto para establecer el valor de la clave primaria

- ♦ **@GeneratedValue (strategy = GenerationType.AUTO):**
  - ♦ AUTO: por defecto, el gestor de persistencia elige. → SEQUENCE
  - ♦ IDENTITY:
    - ♦ fácil de usar pero no el mejor en cuanto a rendimiento
    - ♦ Se basa en el auto incremento de una columna en la BD y esta genera un nuevo valor en cada inserción.
  - ♦ SEQUENCE:
    - ♦ Usa una secuencia de la BD para generar valores únicos
    - ♦ No supone impacto en el rendimiento **pero genera una nueva tabla terminada en \_seq**
  - ♦ TABLE: casi no se usa, es muy lenta.

- ♦ **Anotación:**

- ♦ **@Column:** especifica el mapeo entre un atributo de la clase y una columna de una tabla de la BD.
- ♦ Atributos interesantes:
  - ♦ *name*: permite establecer el nombre de la columna de la tabla
  - ♦ *length*: permite definir la longitud de la columna en caracteres
  - ♦ *nullable*: crea una restricción Not Null para impedir la inserción de valores nulos.
  - ♦ *scale*: utilizada en columnas de tipo decimal, para indicar el número de decimales
  - ♦ *precision*: utilizada en columnas de tipo decimal, para indicar el número de elementos de la parte entera.
  - ♦ *columnDefinition*: permite especificar el tipo de datos SQL, por ejemplo, `columnDefinition = "char"`

- ♦ **Anotaciones Hibernate:**

- ♦ **@Transient:** permite excluir un campo de ser persistente de la base de datos
- ♦ **@Enumerated:** permite definir una propiedad como enumerado
  - ♦ Atributo interesante: EnumType: permite indicar cómo persiste en la BD el enumerado:
    - ♦ ORDINAL: se almacena solo el ordinal del enumerado
    - ♦ STRING: se almacena el nombre

- ♦ **Anotación:**

- ♦ **@OneToOne**: define una relación 1:1 entre dos entidades
  - ♦ **@OneToMany**: define una relación 1:N entre dos entidades
  - ♦ **@ManyToOne**: define una relación N:1 entre dos entidades
  - ♦ **@ManyToMany**: define una relación N:M entre dos entidades
  - ♦ **@JoinColumn**: especifica la columna que actúa como clave foránea en una relación
  - ♦ **@NamedQuery**: declara el nombre de una consulta a una entidad.
  - ♦ **@NamedNativeQuery**: declara una consulta SQL nativa a una entidad
-

# Utilización de anotaciones

- ♦ Entonces, para establecer el mapeo con anotaciones necesitamos:
  - ♦ Crear una clase (Bean) que utilice las anotaciones para el enlazado. Ej.: Modulo
  - ♦ Modificar el archivo: hibernate.cfg.xml para añadir el mapeo, para ello:
    - ♦ Antes del cierre de la etiqueta `</session-factory>` añadiremos:
      - ♦ `<mapping class = "nombre_pkt.nombre_clase"/>`
      - ♦ `<mapping class = "entidades.Modulo"/>`

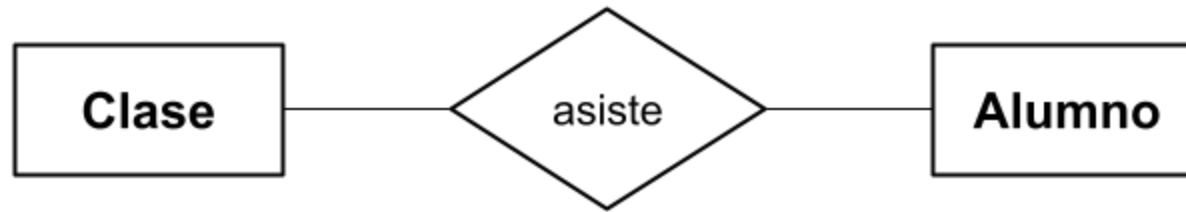


- ♦ A la hora de establecer las relaciones entre las entidades podemos encontrarnos con los siguientes tipos:
    - ♦ Relación 1 a 1
    - ♦ Relación 1 a varios (1:N)
    - ♦ Relación varios a varios (N:M)
-

- ♦ Se utilizará las siguientes anotaciones:
  - ♦ @OneToOne: Indicamos que estableceremos una relación 1 a 1
    - ♦ mappedBy: indica contra qué campo del otro lado de la relación se va a vincular
    - ♦ cascade: permite indicar qué operaciones que afectan a la asociación podrán ser aplicadas en cascada.
    - ♦ orphanRemoval: indica si se quiere aplicar o no la operación de eliminación en aquellas entidades que ya han sido eliminadas de la relación y propagar en cascada su eliminación.
  - ♦ @JoinColumn: nos permitirá crear una clave foránea
    - ♦ name: permite especificar el nombre que tomará la columna dentro de la tabla.
    - ♦ unique, nullable: restricciones en la tabla para impedir valores duplicados (default false). y nulos (default true).
    - ♦ columnDefinition: permite indicar instrucciones SQL sobre cómo va a ser la configuración de una columna.
- ♦ NOTA: @JoinColumn y mappedBy nunca pueden ir en la misma entidad.

# Relaciones: 1:1

- ♦ Existen dos formas de establecer estas relaciones:
  - ♦ Bidireccional ☐ Recomendable
  - ♦ Unidireccional



# Relaciones: 1:1

- ◆ Distinguiamos dos lados de la relación: Propietaria y inverso


- ◆ Propietario

```
@Entity (name = "alumnos")  
public class Alumno {
```

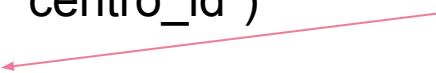
```
@OneToOne  
@JoinColumn(name = "centro_id")  
private Centro centro;
```

```
public void setCentroBidireccional(Centro centro){  
    // Asignar el nuevo centro al alumno  
    this.centro = centro;  
    // Asignar el alumno al centro  
    centro.setAlumno(this);  
}
```

Nombre que  
queramos que  
tenga la columna  
en la tabla de la  
BD



Nombre del atributo de la  
clase donde estará la PK



Asignación  
bidireccional



- ◆ Inverso

```
@Entity(name = "centros")  
public class Centro {
```

```
@OneToOne(mappedBy = "centro")  
private Alumno alumno;
```

```
public void setAlumno(Alumno alumno){  
    this.alumno = alumno;  
}
```

# Relaciones: 1:N

- ♦ Para establecer una relación 1 a N utilizaremos las siguientes anotaciones:
  - ♦ @OneToMany + lista en el lado 1
  - ♦ @ManyToOne + atributo del tipo contrario en el lado N
- ♦ Relación: 1 alumno, varios teléfonos
- ♦ Lado 1: (Alumno)

```
@OneToMany(mappedBy = "alumno")  
private List<Telefono> telefonos;
```

Nombre del atributo de la clase donde  
estará la PK

- ♦ Lado N: (Teléfonos)

```
@ManyToOne  
@JoinColumn(name = "telefono_id")  
private Alumno alumno;
```

Nombre que queramos que tenga la  
columna en la tabla

# Relaciones: N:M

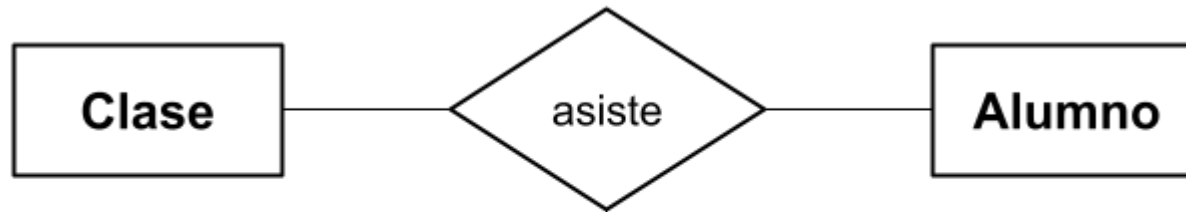
---

- ♦ A la hora de representar una relación varios a varios existen tres formas distintas de implementarla:
    - ♦ Implementación básica
    - ♦ Implementación usando claves compuestas
    - ♦ Implementación creando una nueva entidad
  - ♦ Usar una u otra dependerá de la casuística en particular.
  - ♦ Se optará siempre que se pueda por la implementación básica
-

# Relaciones: N:M → Implementación básica

31

- ♦ Se utilizará las siguientes anotaciones:
  - ♦ @ManyToMany: Indicamos que estableceremos una relación N a M
  - ♦ @JoinTable: para establecer la clave foránea
    - ♦ *name*: permite especificar el nombre de la tabla de asociación.
    - ♦ *joinColumns*: permite indicar el nombre de la columna (en la base de datos) que hace referencia a la tabla propietaria de la relación.
    - ♦ *inverseJoinColumns*: permite indicar el nombre de la columna (en la base de datos) que hace referencia a la tabla no propietaria de la relación.



# Relaciones: N:M → Implementación básica

32

Varios alumnos van a varias clases

- ♦ Lado propietario (Alumno):

```
@ManyToMany
@JoinTable(
    name = "Alumno_Clases",
    joinColumns = @JoinColumn(name = "alumno_id"),
    inverseJoinColumns = @JoinColumn(name = "clase_id")
)
private List<Clase> listaClases;
```

Nombre de la tabla intermedia

Nombre de la columna intermedia  
donde estará la clave de la relación  
propietaria

Nombre de la columna intermedia  
donde estará la clave de la relación  
inversa

- ♦ Lado inverso (Clases):

```
@ManyToMany(mappedBy = "listaClases")
private List<Alumno> listaAlumnos;
```



# Relaciones: N:M → Usando claves compuestas

33

## 1. Se crea una clave clave compuesta:

- ♦ Creamos una nueva clase que se marca como `@Embeddable`
- ♦ La nueva clase implementará la interfaz `Serializable`
- ♦ Un atributo por cada una de los atributos que son clave de la relación

```
@Embeddable  
class AsistePK implements Serializable{  
  
    @Column(name = "clase_id")  
    Long claseId;  
  
    @Column(name = "alumno_id")  
    Long alumnoId;  
}
```

# Relaciones: N:M → Usando claves compuestas

34

## 2. Se crea manualmente la entidad intermedia

- ♦ La marcamos como una entidad
- ♦ Usamos `@EmbeddedId` para indicar que la entidad anterior contiene los IDs
- ♦ Indicamos cómo mapeamos cada ID con su atributo Java con `@MapsId`
- ♦ Creamos las relaciones 1:N con las entidades originales

```
@Entity  
class Asiste {
```

```
    @EmbeddedId  
    AsistePK id;
```

```
    @ManyToOne  
    // Enlazamos la propiedad de la entidad con la propiedad  
    // en Java de la clave compuesta  
    @MapsId("alumnoid")  
    @JoinColumn(name = "alumno_id") // Enlazar con el campo de la BD  
    Alumno alumno;
```

```
    @ManyToOne  
    // Enlazamos la propiedad de la entidad con la propiedad  
    // en Java de la clave compuesta  
    @MapsId("claselid")  
    @JoinColumn(name = "clase_id") // Enlazar con el campo de la BD  
    Clase clase;
```

```
    double nota;  
}
```

# Relaciones: N:M → Usando claves compuestas

35

3. Se asigna relaciones 1:N entre las entidades originarias y la entidad intermedia

```
@Entity
@Table(name = "alumnos")
public class Alumno{

    @OneToMany(mappedBy = "alumno")
    private List<Asiste> listaAsiste;

}
```

```
@Entity
@Table(name = "clases")
public class Clase {

    @OneToMany(mappedBy = "clase")
    private List<Asiste> listaAsiste;

}
```

# Relaciones: N:M → Creando nueva entidad

36

```
@Entity
class Asiste {

    @Id
    Long id;

    @ManyToOne
    Alumno alumno;

    @ManyToOne
    Clase clase;

    double nota;
}
```

```
@Entity
@Table(name = "alumnos")
public class Alumno{

    @OneToMany(mappedBy = "alumno")
    private List<Asiste> listaAsiste;

}
```

```
@Entity
@Table(name = "clases")
public class Clase {

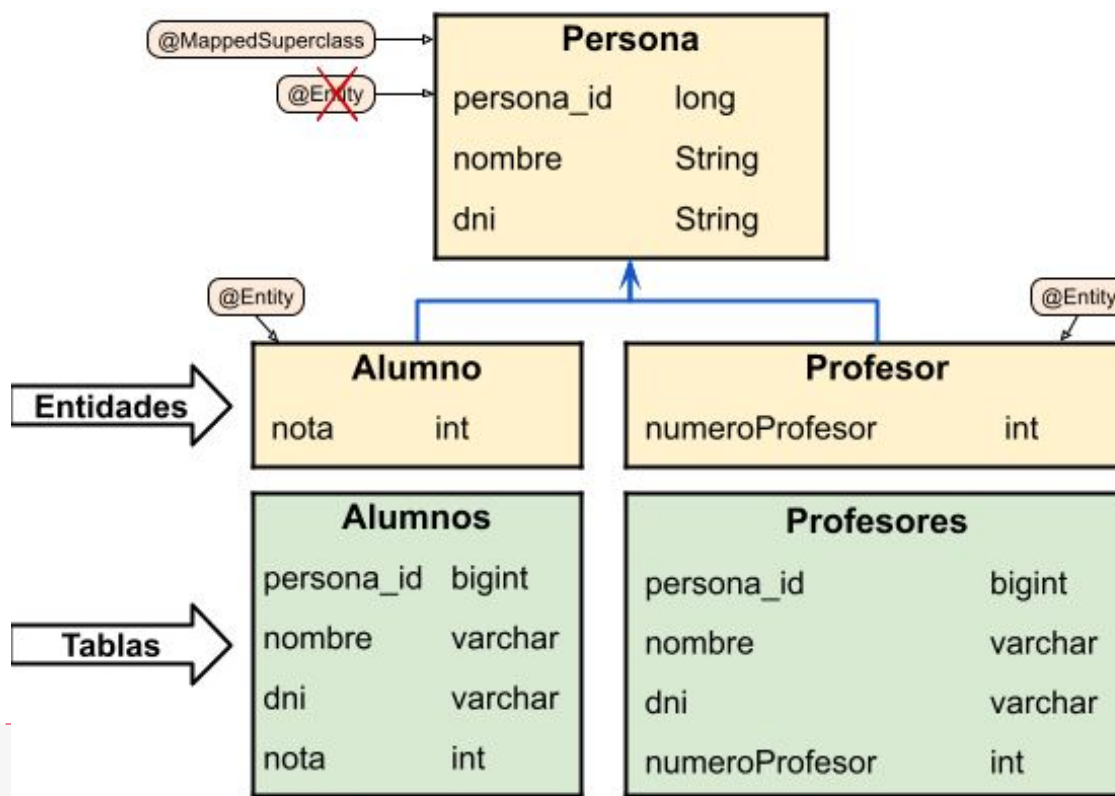
    @OneToMany(mappedBy = "clase")
    private List<Asiste> listaAsiste;

}
```

- ◆ A la hora de replicar la herencia existen 4 opciones:
  - ◆ **MappedSuperclass**
  - ◆ **JoinedTable**
  - ◆ **Tabla por clase**
  - ◆ **Single Table**

# Herencia: MappedSuperclass

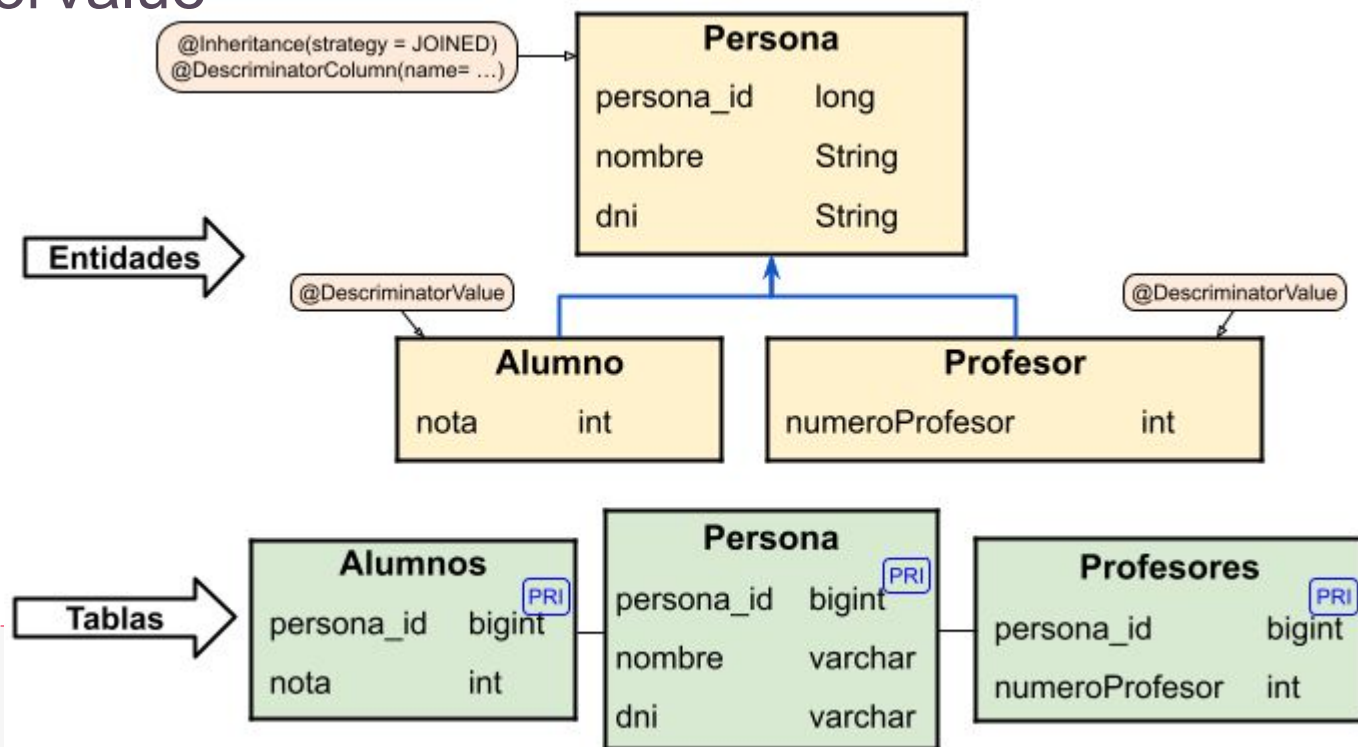
- ♦ “Herencia básica” donde una clase extiende/hereda las propiedades de otra.
- ♦ La clase padre se marcará con `@MappedSuperclass`
- ♦ La clase hijo extenderá la clase padre `extends clasePadre`



# Herencia: JoinedTable

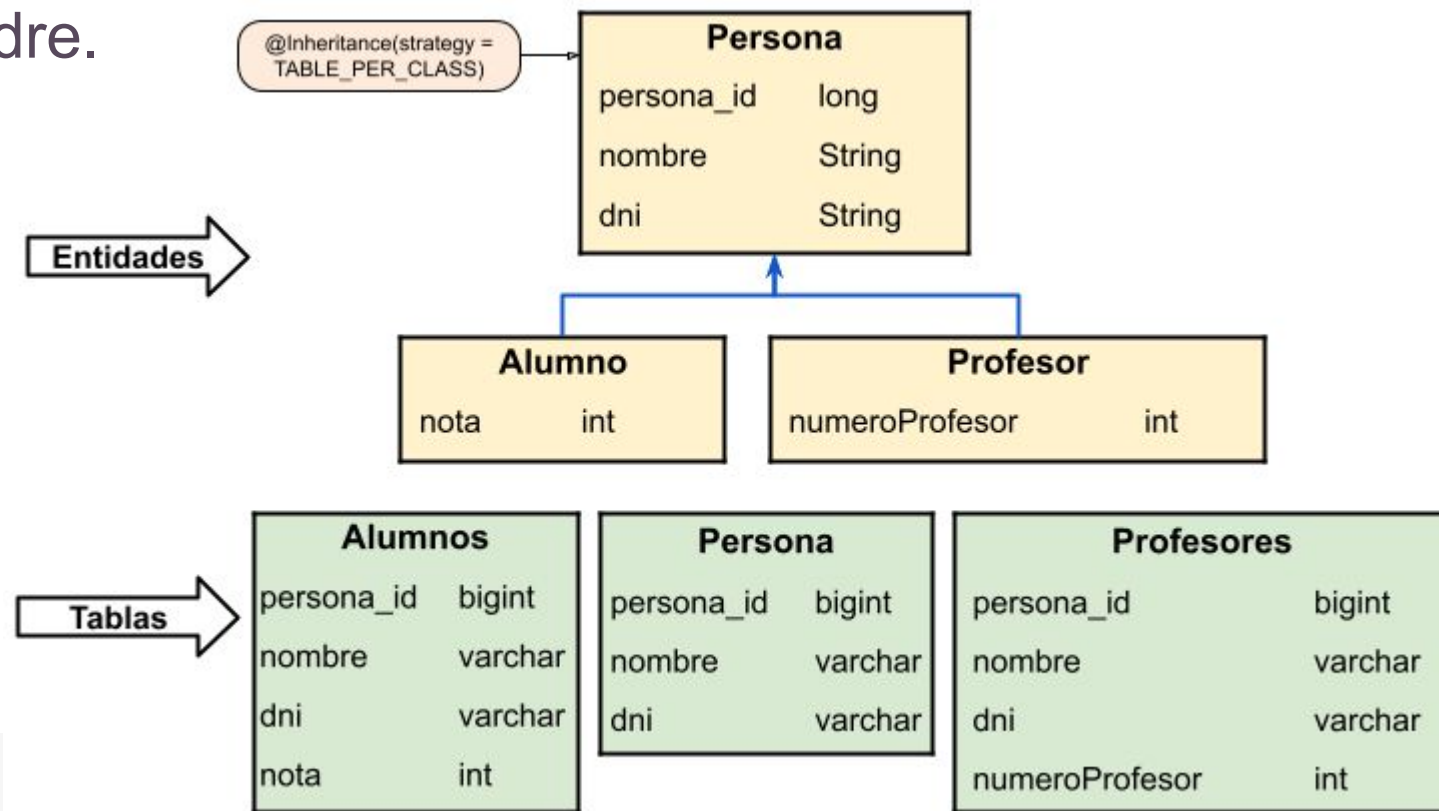
- ♦ Se tiene una tabla por cada clase. □ Tablas relacionadas entre sí por FKs.
- ♦ La clase padre se marcará como □ @Inheritance(strategy = InheritanceType.JOINED)
- ♦ La clase hijo □ extends *clasePadre*

las distinguimos con □ @DiscriminatorValue



# Herencia: Tabla por clase

- ♦ Se tiene una tabla por cada clase. □ Tablas no relacionadas entre sí.
- ♦ La clase padre se define usando □  
`@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`
- ♦ La clase hijo extiende la clase padre.





# Herencia: Single Table

- ♦ Se mapean todas las entidades de la estructura de herencia a la misma tabla de la base de datos.
- ♦ Esta estrategia mejora el rendimiento de las consultas polimórficas.
- ♦ Inconvenientes:
  - ♦ Los atributos de todas las entidades son mapeadas a la misma tabla de la base de datos.
  - ♦ Cada entrada usa un conjunto de columnas y establece el resto a nulo.
  - ♦ Se crean problemas de integridad.
- ♦ Es la estrategia utilizada por defecto en JPA si no se especifica.
- ♦ La clase padre:
  - ♦ Define el identificador □ @Id
  - ♦ Utiliza la siguiente anotación □ @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
- ♦ Clase hijo □ extiende la clase padre

# Herencia: Single Table

- ♦ En la estrategia anterior todos los registros están en la misma tabla □ Necesitamos una forma de distinguir los tipos.
- ♦ Usaremos las siguientes anotaciones:

- ♦ @DiscriminatorColumn y @DiscriminatorValue

- ♦ En la clase padre, indicamos la columna por la que estableceremos la distinción:

@DiscriminatorColumn(name="tipo\_vehiculo", discriminatorType = DiscriminatorType.INTEGER)

- ♦ En la clase hijo, indicamos el valor que tendrá esa columna para cada una de las clases

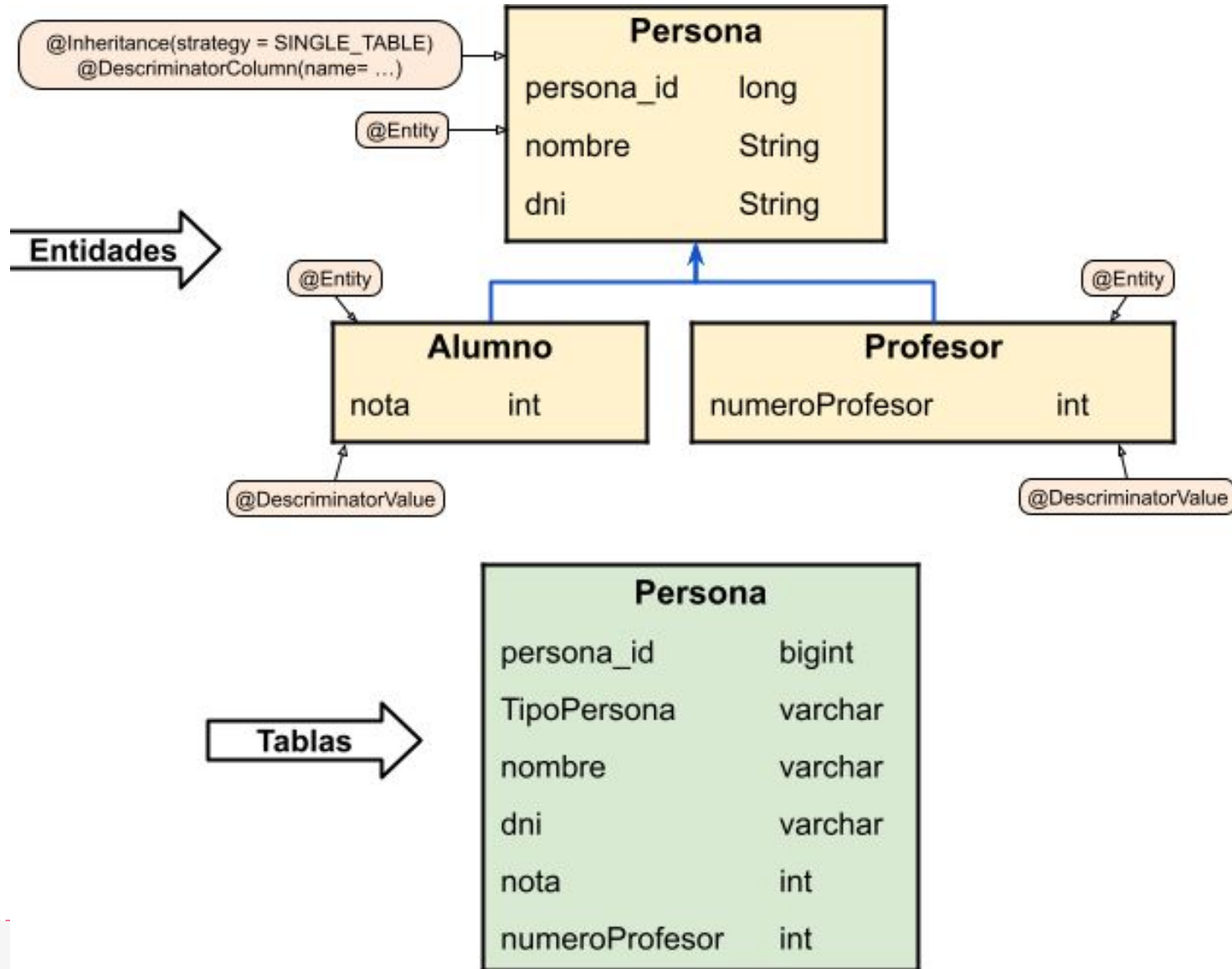
@DiscriminatorValue("1")

- ♦ @DiscriminatorFormula: otra opción es indicar una fórmula en la clase padre que establezca los propios valores de cada clase

@DiscriminatorFormula("case when color is not null then 1 else 2 end")

# Herencia: Single Table

43



# Operaciones con objetos persistentes

- ♦ Almacenar objetos:
  - ♦ `session.persist(objeto)`
- ♦ Cargar objetos:
  - ♦ `session.get(Clase.class, id)` → devuelve nulo si no existe
  - ♦ `session.load(Clase.class, id)` → lanza excepción si no existe
- ♦ Borrar un objeto:
  - ♦ `session.remove(objeto)`
- ♦ Actualizar un objeto:
  - ♦ `session.merge(objeto)`

- ♦ Creación de una interfaz con los métodos a implementar
- ♦ Para cada una de las entidades crear una clase *nombreEntidadRepository* que implemente la interfaz para la creación de los métodos.
- ♦ Todo repositorio de hibernate debería tener:
  - ♦ Una variable privada de tipo Session
  - ♦ Un constructor que reciba el objeto Session con el que trabajar

- ♦ Similar a SQL pero en lugar de hacer referencia a las tablas hace referencia a las clases.

- ♦ La consulta mínima que se puede realizar en HQL es:

- ♦ FROM Ciclo c; => select \* from Ciclo;

- ♦ Diferencias entre HQL y SQL:

- ♦ Ejemplo HQL:

SELECT c FROM Ciclo c ORDER BY nombre

- ♦ Ciclo hace referencia a la clase Java *Ciclo* y NO a la tabla *CicloFormativo*.
  - ♦ Es necesario definir el alias *c* de la clase Java *Ciclo*.
  - ♦ Tras la palabra SELECT se usa el alias en vez del \*.
  - ♦ Al ordenar los objetos se usa la propiedad nombre de la clase *Ciclo* en vez de la columna *nombreCiclo* de la tabla *CicloFormativo*.

- ♦ Parámetros en consulta

- ♦ Opción 1:

```
Dpto depto = (Dpto) session.createQuery("FROM Dpto d where id=" + id).getSingleResult();
```

```
List<Dpto> depto = session.createQuery("FROM Dpto d where id=" + id).getResultList();
```

- ♦ Opción 2 (preferible):

```
Query query = session.createQuery("FROM Emp e WHERE e.dptoElement.id=:id");
```

```
query.setParameter("id", id);
```

```
// List<Emp> empleados = query.getResultList();
```

```
Emp empleado = query.getSingleResult();
```

# Consultas por criterios: List()

- ♦ El método list() nos retorna una lista con todos los objetos que ha retornado la consulta.
- ♦ El método list() puede devolver dos opciones:

- ♦ Una lista de objetos

```
Query query = session.createQuery("SELECT p FROM Profesor p");

List<Profesor> profesores = query.list();

for (Profesor profesor : profesores) {
    System.out.println(profesor.toString())
}
```

- ♦ Una lista de arrays de objetos

```
Query query = session.createQuery("SELECT p.id,p.nombre FROM Profesor p");

List<Object[]> listDatos = query.list();

for (Object[] datos : listDatos){
    System.out.println(datos[0] + "--" + datos[1]);
}
```



# Consultas por criterios: uniqueResult()


- ♦ En muchas ocasiones una consulta únicamente retornará cero o un resultado.
- ♦ En ese caso es poco práctico que nos retorne una lista con un único elemento.
- ♦ Para facilitarnos dicha tarea Hibernate dispone del método **uniqueResult()**.

```
Profesor profesor = (Profesor) session.createQuery("SELECT p FROM Profesor p WHERE  
id=1001").uniqueResult();  
System.out.println("Profesor con Id 1001=" + profesor.getNombre());
```

# Consultas por nombres

- ♦ En la arquitectura SW se recomienda que las consultas no estén escritas en el código si no en un fichero externo para que no se puedan modificar fácilmente.
- ♦ Hibernate permite realizar algo similar de forma fácil. Para ello utiliza las siguientes anotaciones:
  - ♦ @NamedQueries: permite definir un conjunto de consultas
  - ♦ @NamedQuery: permite definir una consultas.
- ♦ Ambas anotaciones se sitúan antes de la definición del nombre de entidad

Nombre de parámetro, también se podría usar ?



```
@Entity
@NamedQueries({
    @NamedQuery(name="Profesores.findAll", query="SELECT p FROM Profesor p")
    @NamedQuery(name="Profesores.ByName", query="SELECT p FROM Profesor p where p.nombre =:nombre and
    p.ape1=:ape1 and p.ape2=:ape2")
})
public class Profesor {
    ...
}
```

- Donde:
  - Name: nombre de la consultas.
  - Query: consultas en formato HQL

# Consultas por nombres

- ♦ Para usar las consultas desde Java

```
1: Query query = session.getNamedQuery("Profesores.findAll");
2: List<Profesor> profesores = query.list();
3: for (Profesor profesor : profesores) {
4:     System.out.println(profesor.toString());
5: }
```

```
1: String nombre="ISIDRO";
2: String ape1="CORTINA";
3: String ape2="GARCIA";
4:
5: Query query = session.getNamedQuery(Profesores.ByName)
6: query.setString("nombre",nombre);
7: query.setString("ape1",ape1);
8: query.setString("ape2",ape2);
9:
10: List<Profesor> profesores = query.list();
11: for (Profesor profesor : profesores) {
12:     System.out.println(profesor.toString());
13: }
```

- ♦ Para la ejecución de consultas SQL se utiliza el método `createSQLQuery()`
- ♦ A pesar de existir esta funcionalidad, no es recomendable abusar de ella y solo recurrir a ella en casos justificados, por ejemplo:
  - ♦ Cuando suponga un aumento del rendimiento importante
  - ♦ Cuando se requiera una ordenación particular de los resultados,
  - ♦ Cuando hay que hacer consultas muy complejas que no es posible hacerlo con el lenguaje HQL.

- ♦ Consulta básica:

```
session.createSQLQuery("SELECT id, nombre, edad from Personas").list();
```

- ♦ Consulta más compleja □ consulta de entidades

```
sess.createSQLQuery("SELECT * FROM PERSONAS").addEntity(Persona.class);
```

# Gestión de transacciones

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Utilizar la Session para saveOrUpdate/get/delete/...tx.commit();
} catch (Exception e) {
    if (tx != null) {
        tx.rollback();
        throw e;
    }
} finally {
    session.close();
} // Al finalizar la aplicación ...HibernateUtil.shutdown( );
```