

Arquitecturas de Software para Aplicaciones Empresariales

Manejo de herencia en JPA y Spring Boot



PROGRAMA DE INGENIERIA DE SISTEMAS

Ing. Daniel Eduardo Paz Perafán (danielp@Unicauca.edu.co)

Ing. Pablo A. Magé (pmage@Unicauca.edu.co)

¿Cómo persistimos información proveniente de una estructura jerárquica en Java?

- Existen 3 estrategias que nos provee JPA para poder mapear éstas estructuras a un modelo relacional de bases de datos.
- SINGLE_TABLE, JOINED y TABLE_PER_CLASS
- Cada estrategia tiene sus ventajas y desventajas dependiendo del diseño a nivel objetos, pero no debería condicionar al domino al momento de insertar o consultar registros.

SINGLE_TABLE

Mapea toda la jerarquía a una sola tabla, con todos los atributos existentes en la superclase y en las subclases. Definimos la estrategia, con la annotation `@Inheritance` en la superclase

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Getter @Setter @AllArgsConstructor @NoArgsConstructor
public class Producto {
    @Id
    private int idProducto;
    private String nombre;
}
```

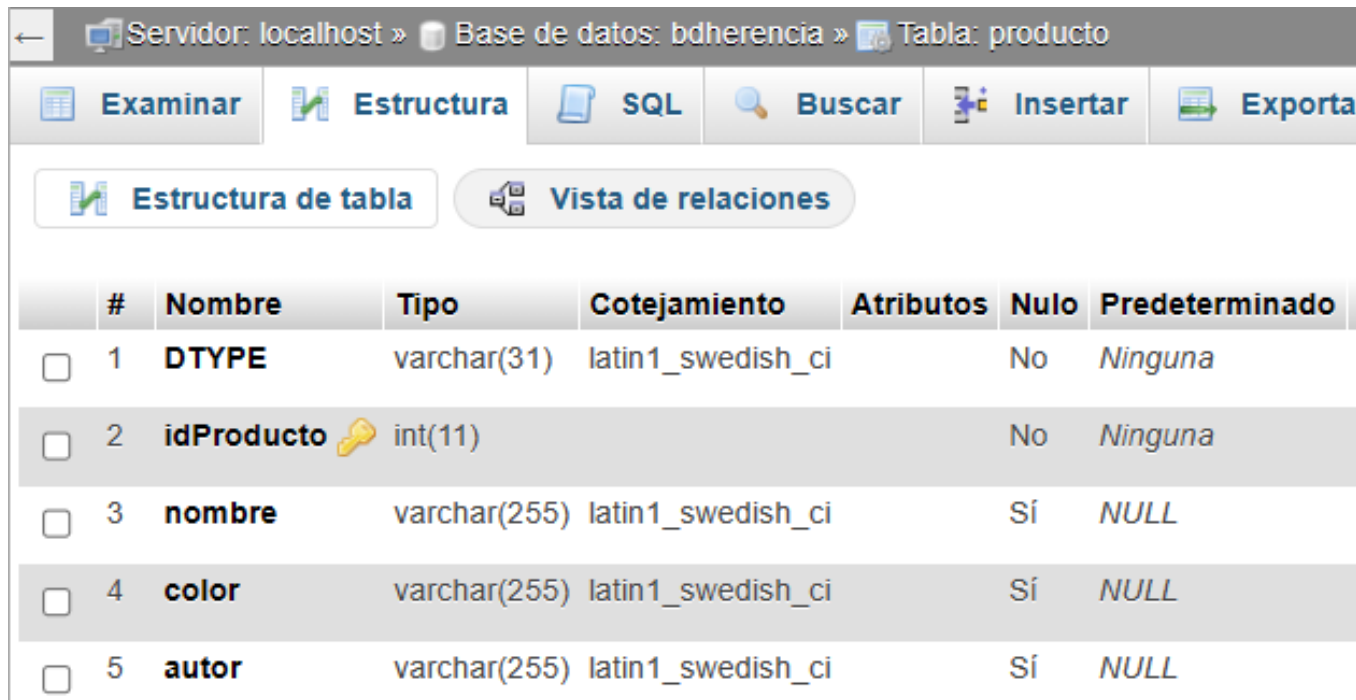
```
@Entity
@Getter @Setter @NoArgsConstructor
public class Lapis extends Producto{
    private String color;

    public Lapis(int idProducto, String nombre,String color)
    {
        super(idProducto, nombre);
        this.color=color;
    }
}
```

```
@Entity
@Getter @Setter @NoArgsConstructor
public class Libro extends Producto{
    private String autor;


    public Libro(int idProducto, String nombre, String autor)
    {
        super(idProducto, nombre);
        this.autor=autor;
    }
}
```

Si consultamos la base de datos, tenemos una sola tabla con el nombre de la superclase, con todos los atributos de todas las clases, y además una columna extra, que representa el campo discriminador para poder identificar a que subclase pertenece el registro.



The screenshot shows a database management interface with the following elements:

- Navigation bar: Servidor: localhost » Base de datos: bdherencia » Tabla: producto
- Buttons: Examinar, Estructura, SQL, Buscar, Insertar, Exportar
- Sub-tabs: Estructura de tabla (selected), Vista de relaciones
- Table structure view:

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
<input type="checkbox"/>	1	DTYPE	varchar(31)	latin1_swedish_ci		No	Ninguna
<input type="checkbox"/>	2	idProducto 	int(11)			No	Ninguna
<input type="checkbox"/>	3	nombre	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/>	4	color	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/>	5	autor	varchar(255)	latin1_swedish_ci		Sí	NULL

Para registrar los datos utilizamos un repositorio de tipo producto

```
private void almacenarLibro() {  
    Libro objLibro = new Libro(idProducto:1, nombre:"Cien años de soledad", autor:"Gabriel Garcia Marquez");  
  
    this.servicioBDProductos.save(objLibro);  
}  
  
private void almacenarLapiz() {  
    Lapiz objLapiz = new Lapiz(idProducto:2, nombre:"Lapiz B1", color:"Negro");  
  
    this.servicioBDProductos.save(objLapiz);  
}
```

La inserción de registros se hace de manera directa sobre la tabla producto, insertando solo los campos correspondientes al tipo de clase que se quiere insertar.

```
and lapiz0_.DTYPE='Lapiz'  
2023-10-20 00:09:26.557 DEBUG 8184 --- [           main] org.hibernate.SQL           : insert into Producto (nombre, autor, DTYPE, idProducto) values  
(?, ?, 'Libro', ?)  
Hibernate: insert into Producto (nombre, autor, DTYPE, idProducto) values (?, ?, 'Libro', ?)  
2023-10-20 00:09:26.588 DEBUG 8184 --- [           main] org.hibernate.SQL           : insert into Producto (nombre, color, DTYPE, idProducto) values  
(?, ?, 'Lapiz', ?)
```

La tabla producto almacena registros de tipo libro y tipo lápiz. Se generan valores null para los campos que no pertenecen a Lápiz y para los que no pertenecen a Libro..






```
SELECT * FROM `producto`
```

☐ Perfilando [[Editar en línea](#)] [[Editar](#)] [[Explicar SQL](#)]

☐ Mostrar todo | Número de filas: 25 ▼ Filtrar filas:

Ordenar según la clave: Ninguna ▼

+ Opciones

				DTYPE	idProducto	nombre	color	autor
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	Libro	1	Cien años de soledad	NULL	Gabriel Garcia Marquez
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	Lápiz	2	Lápiz B1	Negro	NULL

Si la consulta es polimórfica, tenemos la ventaja que se hace directamente sobre la tabla, por ende la velocidad de respuesta puede ser mas rápida que otras estrategias.

```
private void consultarProductos() {  
    Iterable<Producto> listaProductos = this.servicioBDProductos.findAll();  
    for (Producto objProducto : listaProductos) {  
        System.out.println("Id producto: " + objProducto.getIdProducto());  
        System.out.println("Nombre: " + objProducto.getNombre());  
        if (objProducto instanceof Libro) {  
            System.out.println(x:"El producto es un libro");  
            System.out.println("Autor: " + ((Libro) objProducto).getAutor());  
        } else {  
            System.out.println(x:"El producto es un lapiz");  
            System.out.println("Color: " + ((Lapiz) objProducto).getColor());  
        }  
    }  
}
```

```
select producto0_.idProducto as idproduc2_3_, producto0_.nombre as nombre3_3_, producto0_.color as color4_3_, producto0_.autor as  
autor5_3_, producto0_.DTYPE as dtype1_3_ from Producto producto0_
```

Id producto: 1

Nombre: Cien años de soledad

El producto es un libro

Autor: Gabriel Garcia Marquez

Id producto: 2

Nombre: Lapiz B1

El producto es un lapiz

Color: Negro

Cuando la consulta no es polimórfica, tenemos la misma ventaja que la polimórfica, pero la diferencia es que en el WHERE se filtra por el tipo de la clase que se desea obtener. Para realizar la consulta se debe utilizar un repositorio de libros.

```
private void consultarLibros() {  
    Iterable<Libro> listaLibros = this.servicioBDLibros.findAll();  
    for (Libro libro : listaLibros) {  
        System.out.println("Id producto: " + libro.getIdProducto());  
        System.out.println("Nombre: " + libro.getNombre());  
        System.out.println("Autor: " + libro.getAutor());  
    }  
}
```

```
select libro0_.idProducto as idproduc2_3_, libro0_.nombre as nombre3_3_, libro0_.autor as autor5_3_ from Producto libro0_ where  
libro0_.DTYPE='Libro'
```

Id producto: 1

Nombre: Cien años de soledad

Autor: Gabriel Garcia Marquez

JOINED

Genera una tabla por cada clase de la jerarquía, sea abstracta o concreta. Es decir que cada atributo de las clases van a estar en su tabla correspondiente de la base de datos, y mediante JOINS entre las subclases y la superclases se pueden obtener los objetos.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Getter @Setter @NoArgsConstructor @AllArgsConstructor
public class Animal {
    @Id
    private int idAnimal;
    private String especie;

    public Animal(int idAnimal)
    {
        this.idAnimal=idAnimal;
    }
}
```

```
@Entity
@PrimaryKeyJoinColumn(name = "IdPerro")
@Getter @Setter @NoArgsConstructor
public class Perro extends Animal{
    private String nombre;

    public Perro(int idAnimal, String especie,String nombre)
    {
        super(idAnimal, especie);
        this.nombre=nombre;
    }
}
```

Si consultamos la base de datos, tenemos 2 tablas, una que corresponde a la super clase y otra que corresponde a la clase hija. De ésta manera vemos que las tablas son mas fácil de entender a diferencia de SINGLE_TABLE, ya que el esquema queda mejor normalizado y no vamos a tener registros que queden en null.

Servidor: localhost » Base de datos: bdherencia » Tabla: animal

Examinar Estructura SQL Buscar Insertar Exportar

Estructura de tabla Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
<input type="checkbox"/> 1	idAnimal 🔑	int(11)			No	Ninguna
<input type="checkbox"/> 2	especie	varchar(255)	latin1_swedish_ci		Sí	NULL

Servidor: localhost » Base de datos: bdherencia » Tabla: perro

Examinar Estructura SQL Buscar Insertar Exportar

Estructura de tabla Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
<input type="checkbox"/> 1	nombre	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/> 2	IdPerro 🔑	int(11)			No	Ninguna

La tabla de la superclase tiene una relación 'one to one' con las tablas de las subclases, ya que el ID de 'Perro' es a su vez PK de la misma tabla y FK a un registro de la tabla 'Animal'. Esto nos permite hacer el join entre ambas tablas y poder obtener los objetos completos.

Para registrar los datos utilizamos un repositorio de tipo perro

```
private void almacenarPerro() {
    Perro objPerro1 = new Perro(idAnimal:1, especie:"mamifero", nombre:"Pluto");
    this.servicioBDPerros.save(objPerro1);

    Perro objPerro2 = new Perro(idAnimal:2, especie:"mamifero", nombre:"Mateo");
    this.servicioBDPerros.save(objPerro2);
}
```

La inserción de registros se hace sobre la tabla animal y la tabla perro

Para insertar un registro, tenemos como desventaja que necesitamos hacer 2 insert. Primero para insertar en la tabla de la superclase y luego otro insert para la tabla de la subclase.








Si necesitamos hacer insert masivos, ya que vamos a tener el doble de inserciones en la base de datos.

```

Hibernate: insert into Animal (especie, idAnimal) values (?, ?)
2023-10-20 00:43:09.502 DEBUG 9076 --- [          main] org.hibernate.SQL           : insert into Perro (nombr
e, IdPerro) values (?, ?)
Hibernate: insert into Perro (nombre, IdPerro) values (?, ?)
2023-10-20 00:43:09.502 DEBUG 9076 --- [          main] org.hibernate.SQL           : insert into Animal (espec
ie, idAnimal) values (?, ?)
Hibernate: insert into Animal (especie, idAnimal) values (?, ?)
2023-10-20 00:43:09.502 DEBUG 9076 --- [          main] org.hibernate.SQL           : insert into Perro (nombr
e, IdPerro) values (?, ?)
Hibernate: insert into Perro (nombre, IdPerro) values (?, ?)
2023-10-20 00:43:09.702 DEBUG 9076 --- [          main] org.hibernate.SQL           : select perro0_.IdPerro as
```

Los registros se almacenan en dos tablas separadas

+ Opciones

		nombre	IdPerro
<input type="checkbox"/>	 Editar	 Copiar	 Borrar
	Pluto	1	
<input type="checkbox"/>	 Editar	 Copiar	 Borrar
	Mateo	2	

<div><div><div></div><div></div><div></div></div></div>				idAnimal	especie
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	1	mamifero
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	2	mamifero

- De esta forma se cumple con las formas normales
- Admite campos no nulos para cada subclase
- No requiere de un campo discriminador
- Como desventaja se deben realizar insert en diferentes tablas

Si la consulta es no polimórfica, la query obtiene el registro de la subclase y realiza un JOIN contra la tabla que representa la superclase.

```
private void consultarPerros() {  
    Iterable<Perro> listaPerros = this.servicioBDPerros.findAll();  
    for (Perro perro : listaPerros) {  
        System.out.println("Id animal: " + perro.getIdAnimal());  
        System.out.println("Especie: " + perro.getEspecie());  
        System.out.println("Nombre: " + perro.getNombre());  
    }  
}
```

```
select perro0_.IdPerro as idanimal1_0_, perro0_1_.especie as especie2_0_, perro0_.nombre as nombre1_3_ from Perro perro0_ inner join Animal perro0_1_ on perro0_.IdPerro=perro0_1_.idAnimal
```

Id animal: 1

Especie: mamifero

Nombre: Pluto

Id animal: 2

Especie: mamifero

Nombre: Mateo

TABLE_PER_CLASS

Se genera una tabla por cada subclase de la jerarquía, repitiendo los atributos de la superclase, en cada tabla que representan a las subclases.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public abstract class Persona {
    @Id
    private int id;
    private String nombres;
    private String apellidos;
}

@Entity
@Getter
@Setter
public class Empleado extends Persona {
    private String empresa;

    public Empleado() {
        super();
    }

    public Empleado(int id, String nombres, String apellidos, String empresa) {
        super(id, nombres, apellidos);
        this.empresa = empresa;
    }
}

@Entity
@Setter
public class Estudiante extends Persona {
    private float promedio;

    public Estudiante() {
        super();
    }

    public Estudiante(int id, String nombres, String apellidos, float promedio) {
        super(id, nombres, apellidos);
        this.promedio = promedio;
    }
}
```

TABLE_PER_CLASS

El problema es, que al no tener una tabla de referencia que genere los IDs (Como en el caso de la superclase en JOINED), los IDs de los registros en cada tabla se van a repetir y eso nos va a traer un problema al momento de obtener los objetos.

Por tal motivo, tenemos que indicarle al ORM que utilice una estrategia para generar los ID en la superclase, por ejemplo, en la clase abstracta Empleado vamos a tener:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public abstract class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;
    private String nombres;
    private String apellidos;

    public Persona(String nombres, String apellidos) {
        this.nombres = nombres;
        this.apellidos = apellidos;
    }
}
```

TABLE_PER_CLASS

Se crea una tabla donde se almacena la secuencia actual.

The screenshot shows a database management interface with the following elements:

- Navigation Bar:** Includes a back arrow, server path 'Servidor: localhost » Base de datos: bdherencia »', and table name 'Tabla: hibernate_sequence'. It also contains icons for 'Examinar', 'Estructura', 'SQL', 'Buscar', 'Insertar', 'Exportar', and 'Importar'.
- Warning Message:** A yellow box with a warning icon stating: 'La selección actual no contiene una columna única. La edición de la grilla y los enlaces de copiado, e'.
- Status Message:** A green box with a checkmark icon stating: 'Mostrando filas 0 - 0 (total de 1, La consulta tardó 0.0038 segundos.)'.
- SQL Query:** A text area containing the query: `SELECT * FROM `hibernate_sequence``.
- Options:** A checkbox for 'Perfilando' and a link for '[Editar en línea]'.
- Table Controls:** Includes a 'Mostrar todo' checkbox, 'Número de filas:' set to '25', and a 'Filtrar filas:' search box with the placeholder 'Buscar en esta tabla'.
- Table Header:** A dropdown menu labeled '+ Opciones' is open, showing the column 'next_val'.
- Page Number:** The number '5' is displayed at the bottom.

Si consultamos la base de datos, tenemos 2 tablas. Una por cada clase hija, la cual contiene los atributos de la superclase y los atributos de la clase hija.

La creación de las tablas es directa, siendo mas eficiente que la estrategia JOIN, ya que solo crea ambas tablas sin ningún otra relación con la jerarquía.



Servidor: localhost » Base de datos: bdherencia » Tabla: estudiante

Examinar Estructura SQL Buscar Insertar Ex

Estructura de tabla Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
<input type="checkbox"/> 1	id	int(11)			No	Ninguna
<input type="checkbox"/> 2	apellidos	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/> 3	nombres	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/> 4	promedio	float			No	Ninguna



Servidor: localhost » Base de datos: bdherencia » Tabla: empleado

Examinar Estructura SQL Buscar Insertar Ex

Estructura de tabla Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
<input type="checkbox"/> 1	id	int(11)			No	Ninguna
<input type="checkbox"/> 2	apellidos	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/> 3	nombres	varchar(255)	latin1_swedish_ci		Sí	NULL
<input type="checkbox"/> 4	empresa	varchar(255)	latin1_swedish_ci		Sí	NULL

```
private void almacenarEmpleados() {  
    Empleado objEmpleado1 = new Empleado(nombres:"Andres", apellidos:"Perez", empresa:"Exito");  
    Empleado objEmpleado2 = new Empleado(nombres:"Juan", apellidos:"Lopez", empresa:"Jumbo");  
    List<Empleado> listaEmpleados = new LinkedList();  
    listaEmpleados.add(objEmpleado1);  
    listaEmpleados.add(objEmpleado2);  
    this.servicioBDEmpleados.saveAll(listaEmpleados);  
}  
  
private void almacenarEstudiantes() {  
    Estudiante objEstudiane1 = new Estudiante(nombres:"Catalina", apellidos:"Delgado", (float) 5.0);  
    Estudiante objEstudiane2 = new Estudiante(nombres:"Hector", apellidos:"Sanchez", (float) 4.8);  
    List<Estudiante> listaEstudiantes = new LinkedList();  
    listaEstudiantes.add(objEstudiane1);  
    listaEstudiantes.add(objEstudiane2);  
    this.servicioBDestudiantes.saveAll(listaEstudiantes);  
}
```

```
Hibernate: insert into Empleado (apellidos, nombres, empresa, id) values (?, ?, ?, ?)  
2023-10-20 01:40:41.708 DEBUG 3392 --- [main] org.hibernate.SQL : insert into Empleado (ape  
llidos, nombres, empresa, id) values (?, ?, ?, ?)  
Hibernate: insert into Empleado (apellidos, nombres, empresa, id) values (?, ?, ?, ?)  
2023-10-20 01:40:41.708 DEBUG 3392 --- [main] org.hibernate.SQL : insert into Empleado (ape  
rellidos, nombres, empresa, id) values (?, ?, ?, ?)  
Hibernate: insert into Empleado (apellidos, nombres, promedio, id) values (?, ?, ?, ?)  
2023-10-20 01:40:41.798 DEBUG 3392 --- [main] org.hibernate.SQL : insert into Empleado (ape  
rellidos, nombres, promedio, id) values (?, ?, ?, ?)  
Hibernate: insert into Empleado (apellidos, nombres, promedio, id) values (?, ?, ?, ?)  
2023-10-20 01:40:41.802 DEBUG 3392 --- [main] org.hibernate.SQL : select empleado0 .id as i
```

Las consultas no polimórficas, se vuelven mas eficientes ya que se ejecutan sobre una sola tabla.

```
private void consultarEmpleados() {  
    Iterable<Empleado> listaEmpleados = this.servicioBDEmpleados.findAll();  
    for (Empleado objEmpleado : listaEmpleados) {  
        System.out.println("Id: " + objEmpleado.getId());  
        System.out.println("Nombres: " + objEmpleado.getNombres());  
        System.out.println("Apellidos: " + objEmpleado.getApellidos());  
        System.out.println("Empresa donde trabaja: " + objEmpleado.getEmpresa());  
        System.out.println(x:" ---- ---- ----");  
    }  
}
```

```
select empleado0.id as id1_1_, empleado0.apellidos as apellido2_1_, empleado0.nombres as nombres3_1_, empleado0.empresa as empresa4_1_  
from Empleado empleado0_
```

Id: 1

Nombres: Andres

Apellidos: Perez

Empresa donde trabaja: Exito

---- ---- ----

Id: 2

Nombres: Juan

Apellidos: Lopez

Empresa donde trabaja: Jumbo

---- ---- ----

Las consultas no polimórficas, se vuelven mas eficientes ya que se ejecutan sobre una sola tabla.

```
private void consultarEstudiantes() {  
    Iterable<Estudiante> listaEstudiantes = this.servicioBDEstudiantes.findAll();  
    for (Estudiante objEstudiante : listaEstudiantes) {  
        System.out.println("Id: " + objEstudiante.getId());  
        System.out.println("Nombres: " + objEstudiante.getNombres());  
        System.out.println("Apellidos: " + objEstudiante.getApellidos());  
        System.out.println("Promedio: " + objEstudiante.getPromedio());  
        System.out.println(x: " ---- ---- ----");  
    }  
}
```

```
select estudiante0_.id as id1_4_, estudiante0_.apellidos as apellido2_4_, estudiante0_.nombres as nombres3_4_, estudiante0_.promedio as promedio1_2_ fr  
om Estudiante estudiante0_
```

Id: 3

Nombres: Catalina

Apellidos: Delgado

Promedio: 5.0

---- ---- ----

Id: 4

Nombres: Hector

Apellidos: Sanchez

Promedio: 4.8

Las consultas polimórficas, se vuelven menos eficientes ya que requieren hacer una unión entre tablas

```
private void consultarPersonas() {  
    Iterable<Persona> listaPersonas = this.servicioBDPersonas.findAll();  
    for (Persona objPersona : listaPersonas) {  
  
        System.out.println("Id: " + objPersona.getId());  
        System.out.println("Nombres: " + objPersona.getNombres());  
        System.out.println("Apellidos: " + objPersona.getApellidos());  
  
        if (objPersona instanceof Empleado)  
            System.out.println("Empresa donde trabaja: " + ((Empleado) objPersona).getEmpresa());  
        else  
            System.out.println("Promedio: " + ((Estudiante) objPersona).getPromedio());  
  
        System.out.println(x: " ---- -");  
    }  
}
```

Las consultas polimórficas, se vuelven menos eficientes ya que requieren hacer una unión entre tablas

```
select persona0_.id as id1_4_, persona0_.apellidos as apellido2_4_, persona0_.nombres as nombres3_4_, persona0_.empresa as empresa1_1_, persona0_.promedio as promedio1_2_, persona0_.clazz_ as clazz_ from ( select id, apellidos, nombres, empresa, null as promedio, 1 as clazz_ from Empleado union all select id, apellidos, nombres, null as empresa, promedio, 2 as clazz_ from Estudiante ) persona0_
```

Id: 1

Nombres: Andres

Apellidos: Perez

Empresa donde trabaja: Exitosa

Id: 2

Nombres: Juan

Apellidos: Lopez

Empresa donde trabaja: Jumbo

Id: 3

Nombres: Catalina

Apellidos: Delgado

Promedio: 5.0

Id: 4

Nombres: Hector

Apellidos: Sanchez

Promedio: 4.8

COMPARACIÓN ENTRE ESTRATEGIAS

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

Estrategia	A favor	En contra
SINGLE_TABLE	<ul style="list-style-type: none">- Es mas simple- Buena performance en general- Evita generar muchas tablas	<ul style="list-style-type: none">- Los campos no utilizados deben aceptar valores nulos- Puede generar confusión para entender el dominio al almacenar registros de clases hijas en una sola tabla- Necesita un campo discriminador para generar los objetos
JOINED	<ul style="list-style-type: none">- Cumple con las formas normales- Admite campos no nulos para cada subclase- No requiere de un campo discriminador- Soporta todo tipo de relaciones polimórficas	<ul style="list-style-type: none">- Es la estrategia que mas tablas requiere crear- Es la estrategia que mas accesos a la base de datos requiere
TABLE_PER_CLASS	<ul style="list-style-type: none">- Permite campos no nulos para cada subclase- No requiere de un campo discriminador	<ul style="list-style-type: none">- Para consultas polimórficas requiere de uniones que pueden disminuir la performance- Perdemos integridad referencial en relaciones *toOne- Las subclases repiten atributos heredados de la superclase

Muchas gracias
Preguntas

