Tutoriales Curso JPA

Formación Sopra Steria
Formación JPA

Bloque JPA

Versión 1.0 del viernes, 14 de diciembre de 2018

Historial

Versión	Fecha	Origen de la actualización	Redactado por	Validado por
1.0	14/12/2018		Alba Bermejo Solís Adrián Colmena Mateos Emilio Guillem Simón Nicole Tarela Duque Adrián Verdú Correcher Alejandro Mus Mejías	

Índice

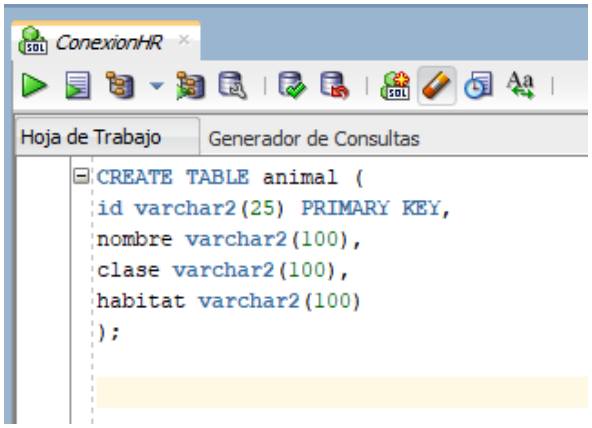
1. Tutorial Capítulo 2

A continuación, se va a introducir a la parte práctica de desarrollar una aplicación Java con JPA. Para ello lo primero será configurar nuestro IDE de programación (Eclipse) para que pueda conectar con nuestra base de datos, en este caso la de Oracle (SQL Developer, HR).

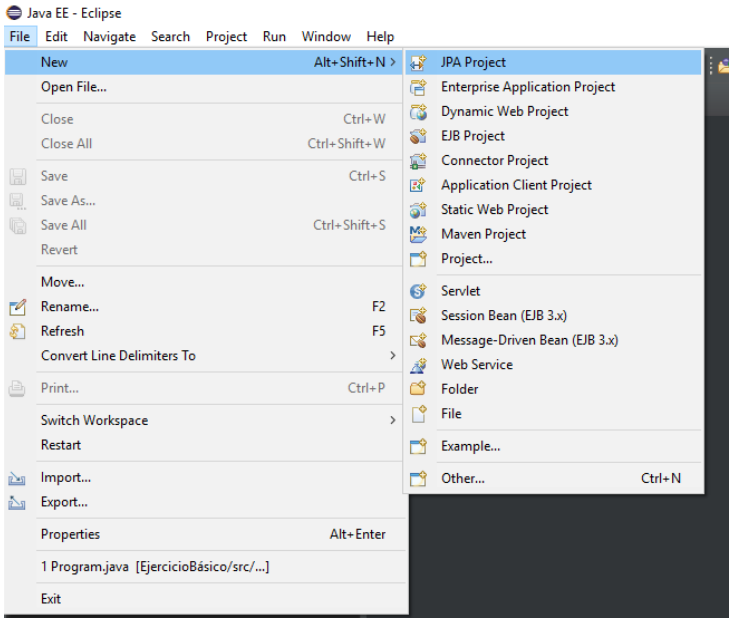
1.1. Configuración inicial

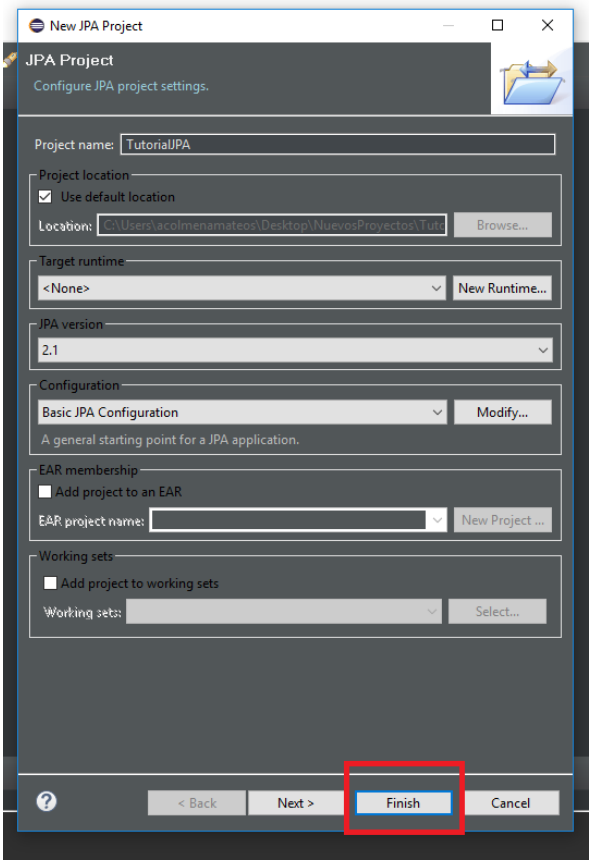
En primer lugar, es necesaria una base de datos (usaremos HR de Oracle) y crear las tablas sobre las que se vaya a trabajar. En este tutorial, solo se va a utilizar una tabla.

A continuación, se crea la tabla *animal* en SQL Developer con los campos id (clave primaria), nombre del animal (en latín), clase y hábitat. Todos los campos serán de tipo varchar2.



Una vez creada la tabla, será necesario conectarla con el IDE Eclipse. En Eclipse, se crea un nuevo proyecto JPA con su archivo Persistence.xml.





Para que el proyecto pueda conectar con base de datos es necesario un driver, en este caso usaremos el de Oracle **ojdbc6.jar**, pudiendo descargarlo en la siguiente URL:

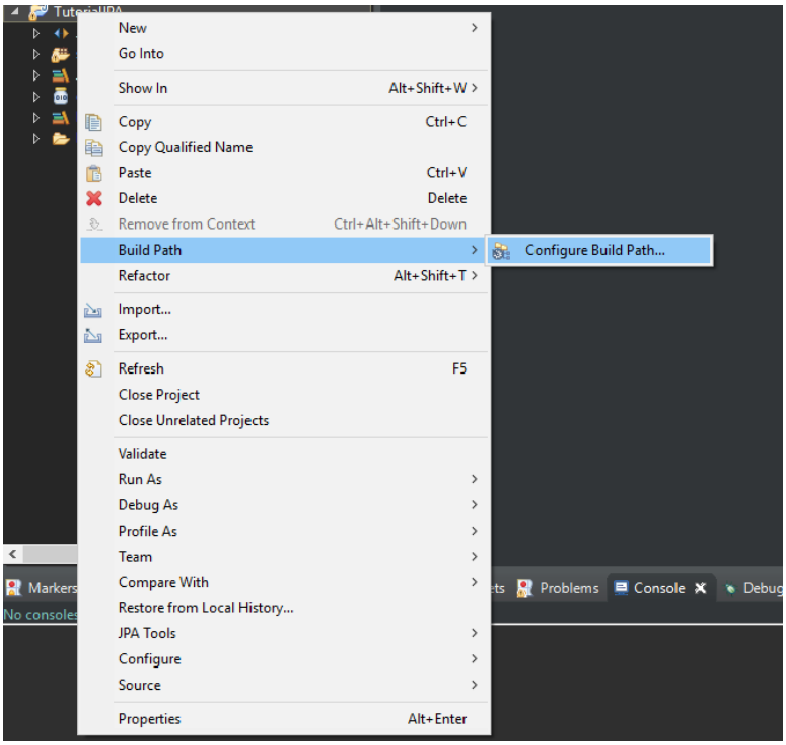
http://www.java2s.com/Code/Jar/o/Downloadojdbc6jar.htm

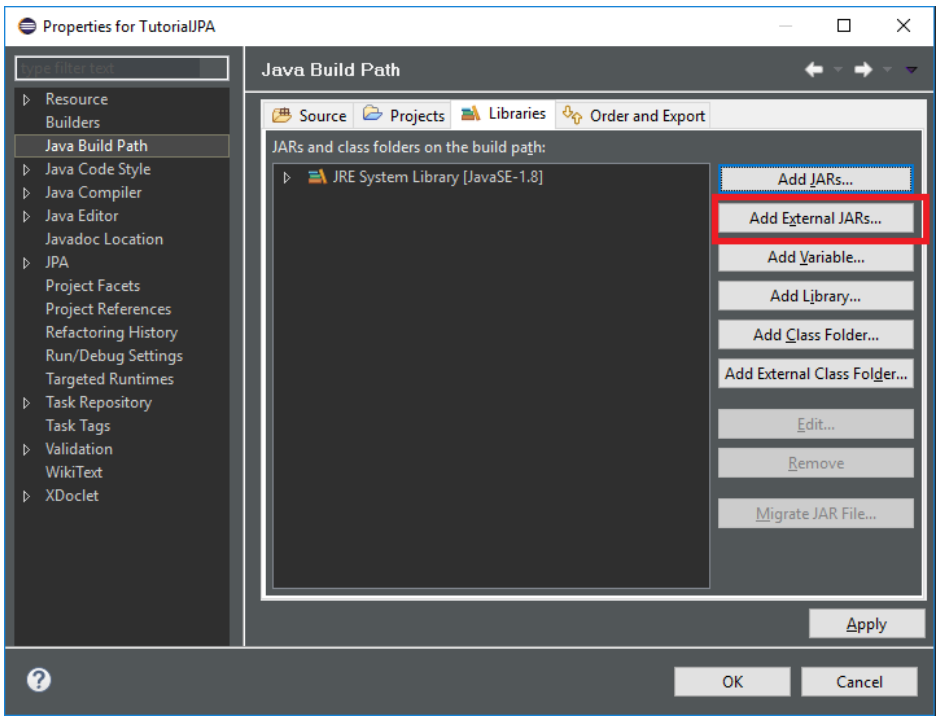
Y descargar las librerías de Hibernate, en esta página:

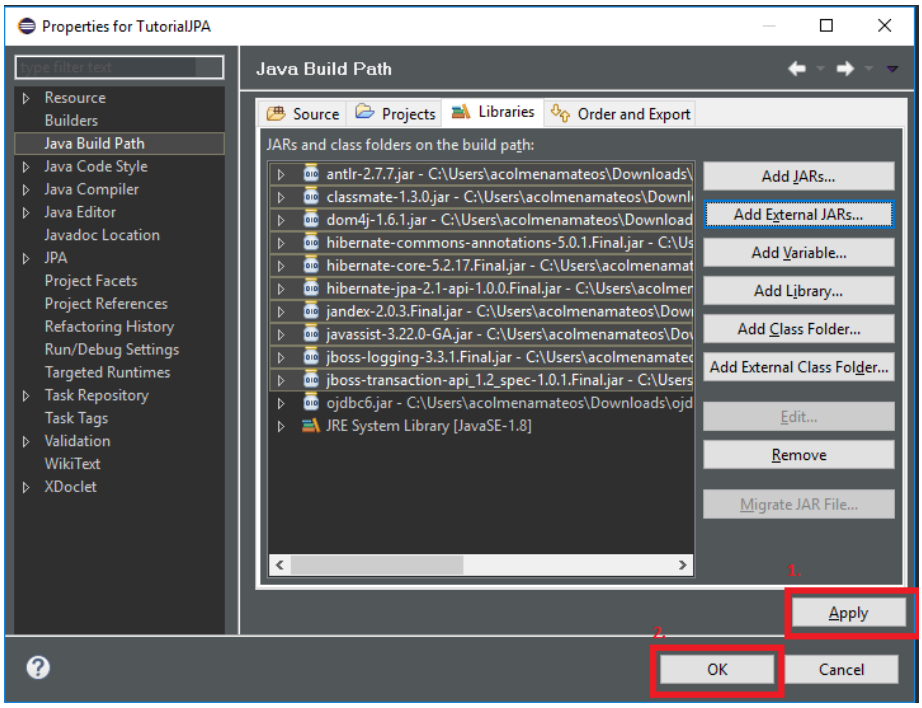
https://sourceforge.net/projects/hibernate/files/hibernate-orm/5.2.17.Final/hibernate-release-5.2.17.Final.zip/download

Hibernate funciona como el *persistence provider* del proyecto JPA creado. No es necesario definirlo en el persistance.xml ya que lo detecta de manera automática una vez que las librerías sean añadidas al proyecto.

Para añadir las librerías de Hibernate, así como el driver de Oracle, es necesario configurar el *path* del proyecto.


New
Go Into
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Validate
Run As
Debug As
Profile As
Team
Compare With
Restore from Local History...
JPA Tools
Configure
Source
Properties Alt+Enter
Configure Build Path...
Markers
Problems
Console


Properties for TutorialJPA
Java Build Path
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
JPA
Project Facets
Project References
Refactoring History
Run/Debug Settings
Targeted Runtimes
Task Repository
Task Tags
Validation
WikiText
XDoclet
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
JRE System Library [JavaSE-1.8]
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
Apply
OK
Cancel



Destacar, que las librerías de Hibernate, se encuentran en lib->required del Zip descargado anteriormente.

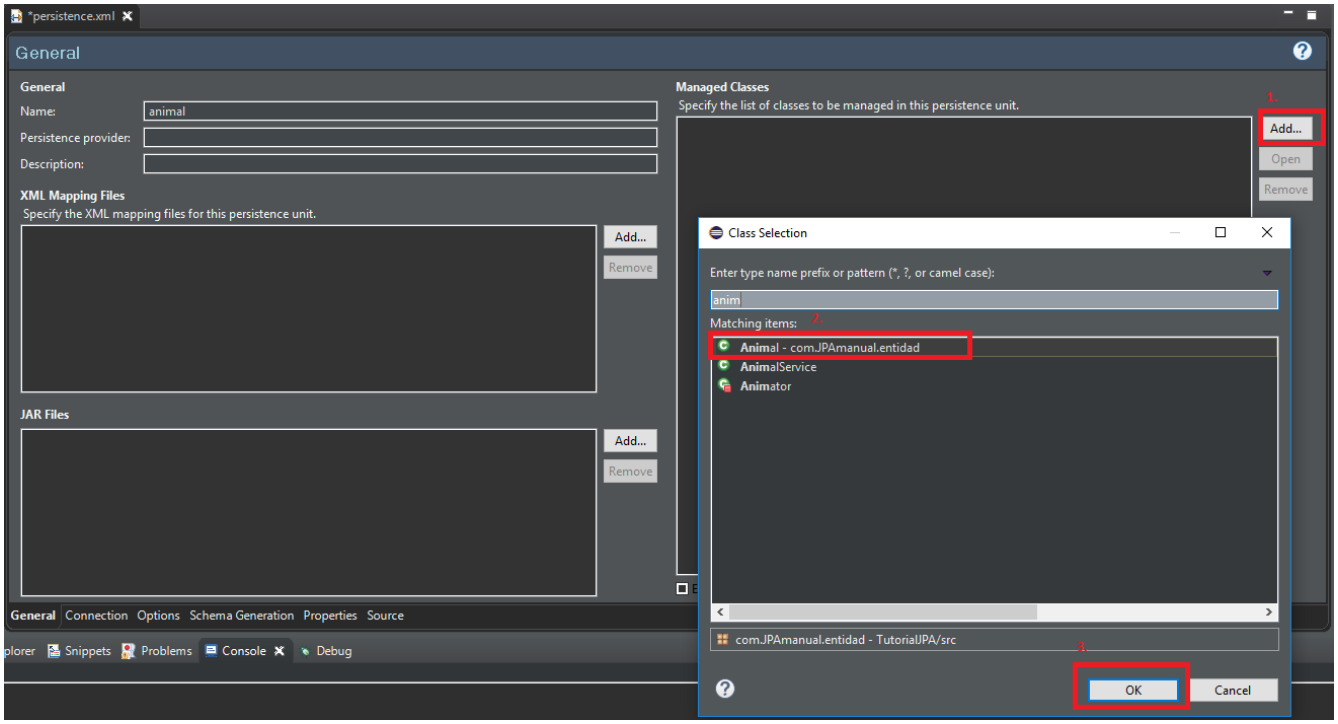
1.2. Creación de una entidad y conexión

Para crear una entidad, es necesario crear un paquete (com.JPAmanual.entidad) y una clase java común (Animal) con la anotación *@Entity*, y la anotación *@Id* para su clave primaria. Es necesario definir los campos de la tabla, a la que hace referencia la entidad, como propiedades de la clase. Añadir también su constructor, un constructor vacío y los métodos *getters* y *setters* de todas las propiedades. La entidad necesita implementar la interfaz *Serializable,* que obliga a sobrescribir (*@Override*) los métodos *equals, hashCode y toString.*

```
package com.JPAmanual.entidad;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
public class Animal implements Serializable {

 private static final long serialVersionUID = 1L;
 @Id
 private String id;
 private String nombre;
 private String clase;
 private String hábitat;
```

```
	public Animal() {
	}

	public Animal(String id, String nombre, String clase, String hábitat) {
		this.id = id;
		this.nombre = nombre;
		this.clase = clase;
		this.hábitat = hábitat;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getClase() {
		return clase;
	}

	public void setClase(String clase) {
		this.clase = clase;
	}

	public String getHábitat() {
		return hábitat;
	}

	public void setHábitat(String hábitat) {
		this.hábitat = hábitat;
	}

	@Override
	public boolean equals(Object elOtro) {
		if (elOtro instanceof Animal) {
			Animal a = (Animal) elOtro;
			return this.id == a.id;
		}
		return false;
	}

	@Override
	public int hashCode() {
		return this.id.hashCode();
	}

	@Override
	public String toString() {
		return String.format("id=%s, nombre=%s, clase=%s, hábitat=%s", this.id,
		this.nombre, this.clase, this.hábitat);
	}
}
```

Por último, es necesario configurar el archivo persistence.xml del proyecto. Dicho archivo debe contener las entidades y la configuración del driver.



Una vez configurado los parámetros anteriores, el código del archivo persistence.xml queda de la siguiente manera:

```
<?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="animal" transaction-type="RESOURCE_LOCAL">
        <class>com.JPAmanual.entidad.Animal</class>
        <properties>
               <property name="javax.persistence.jdbc.url"
value="jdbc:oracle:thin:@localhost:1521:orcl" />
               <property name="javax.persistence.jdbc.user" value="hr" />
               <property name="javax.persistence.jdbc.password" value="hr" />
               <property name="javax.persistence.jdbc.driver"
value="oracle.jdbc.OracleDriver" />

        </properties>
 </persistence-unit>
</persistence>
```

1.3. Creación de servicios de entidad

Se crea un nuevo paquete (com.JPAmanual.servicios) y una nueva clase (ServicioAnimal), donse se crearán los métodos CRUD.

- Método crearAnimal.
  Para crear un animal, es necesario pasarle como parámetros el *EntityManager* (em), y cada una de las propiedades de la clase Animal. Dentro del método creamos un objeto Animal, y luego se lo pasamos al *EntityManager* para que ejecute el método *persist()*.

```
package com.JPAmanual.servicios;

import java.util.ArrayList;
import java.util.List;
import javax.persistence.EntityManager;
import com.JPAmanual.entidad.Animal;

public class ServicioAnimal {

	// Persist
	public static Animal crearAnimal(EntityManager em, String id, String nombre,
	 String clase, String hábitat) {
		Animal a = new Animal(id, nombre, clase, hábitat);
		em.persist(a);
		return a;
}
```

- Método encontrarAnimal.
  Para poder encontrar un animal en concreto de la base de datos y que nos lo devuelva como un objeto de Java se le pasa por parámetros el *EntityManager* (em), y la clave primaria (id). El *EntityManager* localizará al animal con el método *find()*.

```
// Find
public static Animal encontrarAnimal(EntityManager em, String id) {
return em.find(Animal.class, id);
}
```

- Método borrarAnimal.
  Si se quiere borrar un registro de la base de datos desde Java será necesario otra vez pasarle el *EntityManager* (em), y la clave primaria (id), para que encuentre el animal que se quiere borrar. Una que el *EntityManager* lo encuentre, ejecuta el comando *remove()*. Como condición para poder eliminarlo debe haber encontrado un registro con valor disitinto a *null.* Este método devolverá *true* si ha eliminado algún registro de la base de datos y *false*, si el registro no existe o se ha identificado de manera incorrecta.

```
// Remove
public static boolean borrarAnimal(EntityManager em, String id) {
	Animal a = em.find(Animal.class, id);
```

```
        if (a != null) {
                em.remove(a);
                return true;
        }
        return false;
}
```

- Método modificarAnimal.
  Si se quiere actualizar un registro de la base de datos desde Java será necesario otra vez pasarle el *EntityManager* (em), la clave primaria (id), para que encuentre el animal que se quiere modificar. Además, se le debe pasar el resto de propiedades. Una vez que el *EntityManager* lo encuentre, se usan los métodos *set()* de aquellos campos que se quiera cambiar el valor. Este método devolverá el objeto modificado.

```
// Update Animal
public static Animal modificarAnimal(EntityManager em, String id, String nombre,
       String clase, String hábitat) {
        Animal a = em.find(Animal.class, id);
        if (a != null) {
                a.setNombre(nombre);
                a.setClase(clase);
                a.setHábitat(hábitat);
        }
        return a;
}
```

- Método getNombres.
  Si se quiere obtener una lista de los valores de un campo, se utilizan *queries*. Para ello utilizamos la anotación *@NamedQueries*, y la anotación *@NamedQuery* para cada una de las búsquedas que se quieran hacer, en el código de la entidad. Estas *queries* se escribirán en lenguaje JPQL.

```
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
@NamedQueries({ @NamedQuery(name = "Animal.nombre", query = "SELECT a.nombre FROM
Animal a"), })
 public class Animal implements Serializable {
```

  Luego en la clase servicios, se incluirá el método que ejecutará esa *query* en la base de datos y nos devolverá el resultado como un objeto *List.* A dicho método se le debe pasar el *EntityManager* como parámetro, el cual utiliza el método *createNamedQuery(*"Nombre de la query", Clase de la propiedad *).getResultList().*

```
public static List<String> getNombres(EntityManager em) {
        List<String> Nombres = new ArrayList<String>();
        Nombres = em.createNamedQuery("Animal.nombre", String.class).getResultList();
        return Nombres;

}
```

1.4. El método main()

Se crea una última clase (Programa) dentro de un nuevo paquete (com.JPAmanual.main).

Este método será el encargado de ejecutar todos los métodos vistos en la sección anterior. Para ello, se deben crear la *EntityManagerFactory* y los *EntityManger* que se requieran. El objeto *EntityManagerFactory* debe contener el nombre de persistence unit definida en el fichero *persistence.xml*, a la cual se le ha dado el nombre de la tabla en base de datos (animal).

Además, para la correcta ejecución de los métodos CRUD, es necesario abrir una transacción antes de la ejecución de estos. Una vez ejecutados todos los métodos se deberá hacer el *commit* para que los cambios se reflejen en base de datos y la transacción se cierre y cerrar los *EntityManager* y la *EntityManagerFactory.*

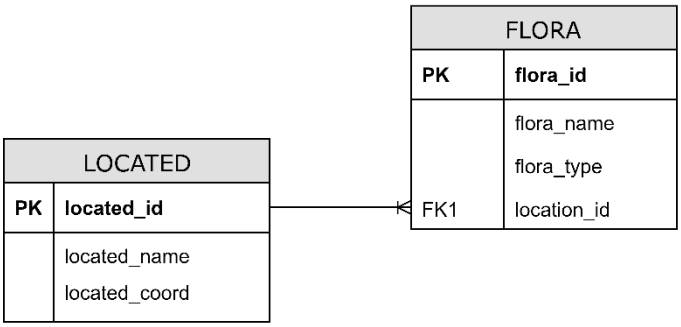
```
package com.JPAmanual.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.JPAmanual.entidad.Animal;
import com.JPAmanual.servicios.ServicioAnimal;


public class Programa {
       public static void main(String[] args){
         EntityManagerFactory emf = Persistence.createEntityManagerFactory("animal");
         EntityManager em = emf.createEntityManager();
         em.getTransaction().begin();

         Animal Hipo = ServicioAnimal.crearAnimal(em, "Hippo", "Hippopotamus
amphibius",
        "Mamífero", "Sabana");

         Animal encontrado= ServicioAnimal.encontrarAnimal(em, "Lion");
         System.out.println(encontrado);

         if(ServicioAnimal.borrarAnimal(em, "Sparrow"))
               System.out.println("Animal eliminado”);
         else
               System.out.println("Error al eliminar");

         ServicioAnimal.modificarAnimal(em, "Hippo", "Hipopótamo", "Herbívoro",
"África");

         System.out.println(ServicioAnimal.getNombres(em));


        em.getTransaction().commit();
        em.close();
        emf.close();
     }
}
```

2. Tutorial Capítulo 3

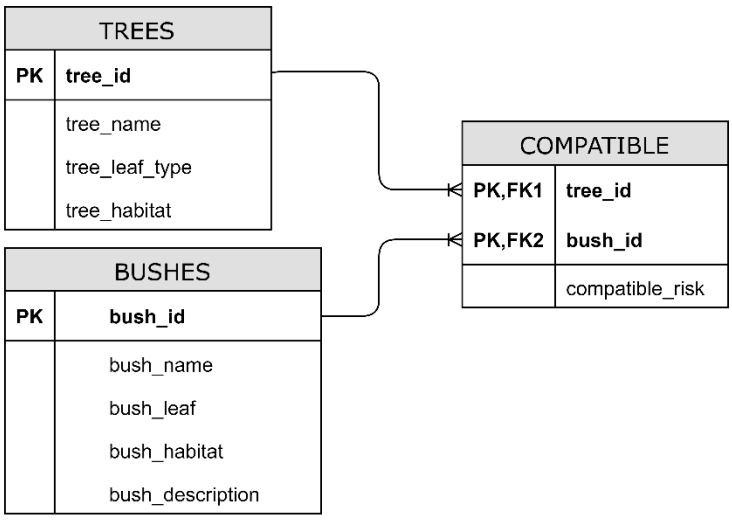
Para revisar el contenido del capítulo sobre el ORM, en este tutorial se presenta el siguiente proyecto.

Se quiere establecer una aplicación para la recogida y evaluación de los datos acerca de la flora de los parques y bosques de una ciudad. La finalidad es de hacer un seguimiento de posibles especies invasoras y evaluar el impacto ambiental.

Por un lado, tenemos un esquema para la toma de datos, como sigue.



En él, se registrarán las coordenadas, el nombre del parque visitado, y según se vayan encontrando especies nuevas en ese parque, se especificará la identificación según un diccionario (flora_type_id) y el tipo (flora_type {'TREES', 'BUSHES'}).



La base de datos consta, por otro lado, con otro esquema de las tablas diccionario de las especies de árboles y arbustos conocidos, así como, la compatibilidad entre ellos, midiendo mediante un marcador (sobre 10) el riesgo que hay de que una especie extinga a la otra.

Ambos esquemas son independientes, aunque para este ejercicio se crearán en la misma base de datos.

2.1. Estableciendo el esquema diccionario

Se crea en SQLDeveloper el esquema anterior, anticipando las primeras modificaciones:

- Las tablas auxiliares *Leaf_type* y *Habitat* sustituirán a los campos análogos y serán comunes para las posibles ampliaciones de nuevas tablas del diccionario de especies. (Nótese que el nombre de los campos existentes en el esquema inicial se mantiene)
- Por simplicidad, la tabla *Compatibles* tiene ahora un identificador único que sustituye a la clave primaria compuesta. (se degradan la restricción *UNIQUE* de las relaciones de la clave primaria compuesta y se suponen privilegios de solo lectura sobre el esquema)

```
CREATE TABLE Leaf_type(
leaf_id INT PRIMARY KEY,
leaf_name VARCHAR2(32)
);

CREATE TABLE Habitat(
habitat_id INT PRIMARY KEY,
habitat_name VARCHAR2(32)
);

CREATE TABLE Trees(
tree_id INT PRIMARY KEY,
tree_name VARCHAR2(64),
leaf_type INT REFERENCES Leaf_type(leaf_id),
habitat INT REFERENCES Habitat(habitat_id)
);

CREATE TABLE Bushes(
bush_id INT PRIMARY KEY,
bush_name VARCHAR2(64),
leaf_id INT REFERENCES Leaf_type(leaf_id),
habitat_id INT REFERENCES Habitat(habitat_id),
bush_description CLOB
);

CREATE TABLE Compatibles(
compatible_id INT PRIMARY KEY,
tree_id INT REFERENCES Trees(tree_id),
bush_id INT REFERENCES Bushes(bush_id),
compatible_risk INT
);


INSERT INTO Leaf_type values (1, 'PERENNE');
INSERT INTO Leaf_type values (2, 'CADUCA');

INSERT INTO Habitat values (1, 'TROPICAL');
INSERT INTO Habitat values (2, 'HUMEDO');
INSERT INTO Habitat values (3, 'SECO');
INSERT INTO Habitat values (4, 'CALIDO');

INSERT INTO Trees values (1, 'Abies pinsapo',1, 1);
INSERT INTO Trees values (2, 'Acer campestre',2, 2);
INSERT INTO Trees values (3, 'Betula pendula',2, 2);
INSERT INTO Trees values (4, 'Ceratonia siliqua',2, 3);
INSERT INTO Trees values (5, 'Citrus aurantium',1, 4);
```

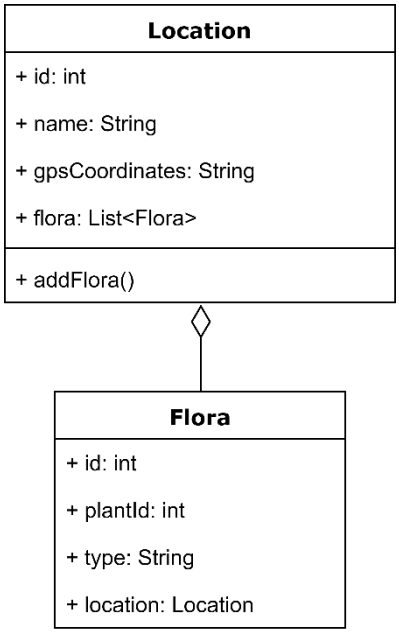
```
INSERT INTO Bushes values (1, 'Magnolia grandiflora',1, 2, 'importada, mayormente uso
ornamental');
INSERT INTO Bushes values (2, 'Convolvulus arvensis',2, 2, 'trepadora, muy
invasiva');
INSERT INTO Bushes values (3, 'Echium creticum',2, 2, 'ornamental');
INSERT INTO Bushes values (4, 'Anchusa azurea',2, 3, 'propiedades medicinales');

INSERT INTO Compatibles values (1, 1, 1, 3);
INSERT INTO Compatibles values (2, 1, 2, 9);
INSERT INTO Compatibles values (3, 1, 3, 1);
INSERT INTO Compatibles values (4, 1, 4, 6);
INSERT INTO Compatibles values (5, 2, 1, 3);
INSERT INTO Compatibles values (6, 2, 2, 8);
INSERT INTO Compatibles values (7, 2, 3, 3);
INSERT INTO Compatibles values (8, 2, 4, 5);
INSERT INTO Compatibles values (9, 3, 1, 2);
INSERT INTO Compatibles values (10, 3, 2, 7);
INSERT INTO Compatibles values (11, 3, 3, 3);
INSERT INTO Compatibles values (12, 3, 4, 5);
INSERT INTO Compatibles values (13, 4, 1, 3);
INSERT INTO Compatibles values (14, 4, 2, 9);
INSERT INTO Compatibles values (15, 4, 3, 3);
INSERT INTO Compatibles values (16, 4, 4, 3);
INSERT INTO Compatibles values (17, 5, 1, 1);
INSERT INTO Compatibles values (18, 5, 2, 7);
INSERT INTO Compatibles values (19, 5, 3, 0);
INSERT INTO Compatibles values (20, 5, 4, 2);
```

2.2. Mapeado de entidades y relaciones

Se van a crear las Tablas *LOCATED* y *FLORA* en la base de datos siguiendo el diagrama de la introducción:

```
CREATE TABLE Located(
located_id INT PRIMARY KEY,
located_name VARCHAR2(64),
located_coord VARCHAR2(32)
);

CREATE TABLE Flora(
flora_id INT PRIMARY KEY,
flora_type_id INT,
flora_type VARCHAR2(32),
located_id INT REFERENCES Located(located_id)
);
```

El diagrama de las clases Java que se persigue es el siguiente:



Para ello se definirán las entidades que relacionan el modelo de dominio con el modelo físico. Primero se plantean las clases *Location* y *Flora* para el compilador,

```
public class Location {

	private int id;
	private String name;
	private String gpsCoordinates;
	private List<Flora> flora;
	. . .
}

public class Flora {

	private int id;
	private int plantId;
	private String type;
	private Location location;
	. . .
}
```

añadiendo después las anotaciones para la definición de la entidad que serán leídas por el proveedor de persistencia.

El código anotado queda de la siguiente manera:

```
@Entity
@Table(name = "LOCATED")
public class Location {

	@Id
	@SequenceGenerator(name = "Gen", sequenceName = "Seq")
	@GeneratedValue(generator = "Gen")
	@Column(name = "LOCATED_ID")
	private int id;
	@Column(name = "LOCATED_NAME")
	private String name;
	@Column(name = "LOCATED_COORD")
	private String gpsCoordinates;
	@OneToMany(mappedBy = "location")
	private List<Flora> flora;

	. . .

}
```

```
@Entity
@Table(name = "FLORA")
public class Flora {

	@Id
	@Column(name = "FLORA_ID")
	@SequenceGenerator(name = "Gen2", sequenceName = "Seq2")
	@GeneratedValue(generator = "Gen2")
	private int id;
	@Column(name = "FLORA_TYPE_ID")
	private int plantId;
	@Column(name = "FLORA_TYPE")
	private String type;
	@ManyToOne
	@JoinColumn(name = "LOCATED_ID")
	private Location location;

	. . .

}
```

Cabe destacar la importancia del papel que juegan las siguientes anotaciones:

@Table

@Column

@SequenceGenerator y *@GeneratedValue*

@ManyToOne y *@JoinColumn*

@OneToMany(mappedBy = "location")

En este punto, si alguna fuera desconocida se recomienda revisarlo en la teoría.

Antes de concluir, dado que se han usado secuencias, estas se deben crear en el modelo físico.

```
CREATE SEQUENCE Seq
MINVALUE 1
START WITH 1
INCREMENT BY 50;
```

```
CREATE SEQUENCE Seq2
MINVALUE 1
START WITH 1
INCREMENT BY 50;
```

2.3. Inserción de datos

Se va a crear un método *main()* para la simulación del trabajo de campo en la inserción de los datos Java. Para ello se crearán la clase estática *Service* y una breve rutina de ejecución como siguen:

```
public class Service {

	. . .

	public static void insertarLocation(EntityManager em, Location location) {
		em.persist(location);
	}

	public static void insertarFlora(EntityManager em, Flora flora) {
		em.persist(flora);
	}

}
```

```
public class Program {

	public static void main(String[] args) {
		EntityManagerFactory emf =
		Persistence.createEntityManagerFactory("XXXX");

		EntityManager em1 = emf.createEntityManager();
		EntityManager em2 = emf.createEntityManager();

		em1.getTransaction().begin();

		Location l1 = new Location();
		l1.setName("Parque natural Sierra Calderona");
		l1.setGpsCoordinates("6515, 1398");

		Location l2 = new Location();
		l2.setName("Parque Natural Sierras de Cazorla");
		l2.setGpsCoordinates("1843, 4915");

		Location l3 = new Location();
```

```
		l3.setName("Parque Nacional de Los Picos de Europa");
		l3.setGpsCoordinates("7337, 9318");

		Service.insertarLocation(em1, l1);
		Service.insertarLocation(em1, l2);
		Service.insertarLocation(em1, l3);

		em1.getTransaction().commit();
		em1.close();


		em2.getTransaction().begin();
		List<Flora> flora = new ArrayList<Flora>() {{
			add(new Flora());
			add(new Flora());
			add(new Flora());
			add(new Flora());
			add(new Flora());
		}};

		Location l = Service.buscarLocation(em2, 1);

		flora.get(0).setType("TREE");
		flora.get(0).setPlantId(1);
		flora.get(0).setLocation(l);

		flora.get(1).setType("BUSH");
		flora.get(1).setPlantId(2);
		flora.get(1).setLocation(l);

		flora.get(2).setType("BUSH");
		flora.get(2).setPlantId(1);
		flora.get(2).setLocation(l);

		flora.get(3).setType("TREE");
		flora.get(3).setPlantId(4);
		flora.get(3).setLocation(l);

		flora.get(4).setType("TREE");
		flora.get(4).setPlantId(2);
		flora.get(4).setLocation(l);

		for (Flora f : flora) {
			Service.insertarFlora(em2, f);
		}

		em2.getTransaction().commit();
		em2.close();

		em.close();
		emf.close();
	}
}
```

2.4. Mapeado de objetos embebidos

Se deberán mapear las entidades *Tree* y *Bush.*

En este caso, para el modelo de dominio Java no tiene sentido el repetir los atributos *leafType* y habitat siendo ambos el mismo grupo de propiedades de cada planta. Es por ello que se creará una clase *Properties* anotada con *@Embeddable* para describir los objetos embebidos asociados a las entidades que mapeen plantas.

```
@Embeddable
@Access(AccessType.FIELD)
public class Properties {

	private Leaf leafType;
	private Habitat habitat;

	public Properties() {
	}

	@Override
	public String toString() {
		return "Leaf: " + leafType + ", Habitat: " + habitat ;
	}

}
```

Los atributos de este objeto serán dos *Enum*: *Leaf* { *NSNC*, *PERENNE*, *CADUCA* } y *Habitat* { *TROPICAL*, *HUMEDO*, *SECO*, *CALIDO* }.

```
public enum Habitat {
	TROPICAL, HUMEDO, SECO, CALIDO
}
```

```
public enum Leaf {
	NSNC, PERENNE, CADUCA
}
```

Para embeber el objeto en las entidades, el atributo *Properties* se mapeará por medio de la anotación *@Embedded* en cada clase. Para que esto funcione, hay que tener en cuenta la diferencia entre los nombres de los atributos y los nombres de los campos en la base de datos introduciendo los elementos de mapeo que aparecen en la anotación *([…] @Column(name = "LEAF_TYPE") […])*.

```
@Entity
@Table(name = "TREES")
public class Tree {

	@Id
	@Column(name = "TREE_ID")
	private int id;
	@Column(name = "TREE_NAME")
	private String name;
```

```
        @Embedded
        @AttributeOverrides({
        @AttributeOverride(
        name = "leafType", column = @Column(name = "LEAF_TYPE")),
        @AttributeOverride(name = "habitat", column = @Column(name = "HABITAT"))
        })
        private Properties properties;

        . . .

}
```

```
@Entity
@Table(name = "BUSHES")
public class Bush {

        @Id
        @Column(name = "BUSH_ID")
        private int id;
        @Column(name = "BUSH_NAME")
        private String name;
        @Embedded
        @AttributeOverrides({
            @AttributeOverride(name = "leafType", column = @Column(name = "LEAF_ID")),
            @AttributeOverride(name = "habitat", column = @Column(name = "HABITAT_ID"))
        })
        private Properties properties;
        @Basic(fetch=FetchType.LAZY)
        @Column(name = "BUSH_DESCRIPTION")
        private String description;

        . . .

}
```

En este apartado se han tratado las siguientes anotaciones:

@Embedded

@AttributeOverrides

@Embeddable

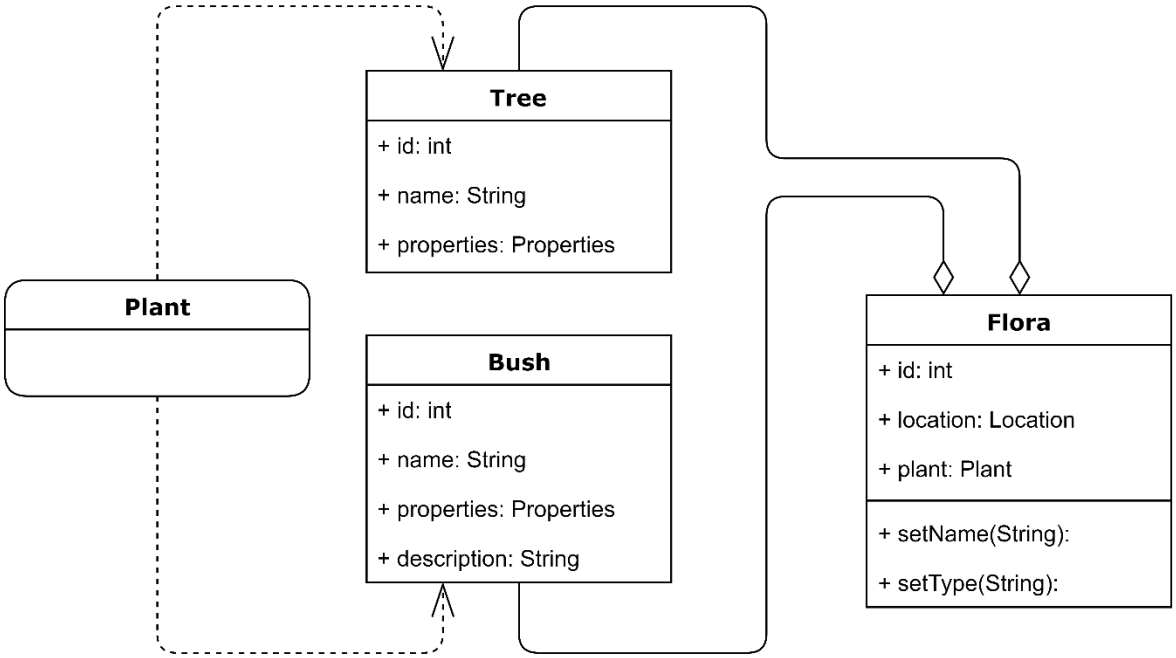
@Access

@Basic(fetch=FetchType.LAZY)

Si fuese necesario, revisar estas anotaciones en la teoría para entender su rol en el mapeo.

2.5. Mapeado de asociaciones, *mixed access* y *transient*

Se va a modificar la clase *Flora* para que contemple una asociación de agregación con las clases *Tree* y *Bush*. Para ello, y dado que a nivel lógico sólo va a contener un único objeto, se creará una clase abstracta *Planta* vacía y se modificarán las clases *Tree* y *Bush* para que ambas extiendan a la clase *Planta*.



De esta manera *Flora* contendrá ahora un objeto de tipo *Planta* que se cargará a través de la propiedad *typeId* de la entidad (accessType.PROPERTY método *getName(String name)*).

Para llevar a cabo esta transformación se realizará un *mixed Access* en la entidad *Flora*.

```
@Entity
@Table(name = "FLORA")
@Access(AccessType.FIELD)
public class Flora {

	@Id
	@Column(name = "FLORA_ID")
	private int id;
	@ManyToOne
	@JoinColumn(name = "LOCATED_ID")
	private Location location;
	@Transient
	private Plant plant;

	public Flora() {
	}

	. . .

	@Access(AccessType.PROPERTY)
	@Column(name = "FLORA_TYPE")
	public String getType() {
		return plant.getClass().toString();
	}
```

```
	public void setType(String type) {
		if (this.plant == null) {
			initPlant(type);
		} else {
			modifyPlant(type);
		}
	}

	. . .

	@Access(AccessType.PROPERTY)
	@Column(name = "FLORA_TYPE_ID")
	public int getTypeId() {
		if (plant instanceof Tree) {
			return ((Tree) plant).getId();
		}
		if (plant instanceof Bush) {
			return ((Bush) plant).getId();

		} else
			return 0;
	}

	public void setTypeId(int id) {
		if (plant == null)
			this.plant = new Tree(id);
		else
			this.plant.setId(id);
	}

	. . .

	//--- métodos auxiliares...

}
```

Dado que se desconoce el orden de carga de las propiedades (*typeId* y *type*) se necesitarán unos métodos auxiliares para cargar correctamente el atributo *plant*.

```
	//--- métodos auxiliares para cargar correctamente la Planta

	 private void initPlant(String type) {
		if (type.equals("TREE"))
			this.plant = new Tree();
		if (type.equals("BUSH"))
			this.plant = new Bush();
	}

	private void modifyPlant(String type) {
		if (type.equals("TREE")) {
			this.plant = new Tree(this.plant.getId());
		} else if (type.equals("BUSH")) {
			this.plant = new Bush(this.plant.getId());
		} else {
			System.err.println("Tipo " + type + " no reconocido");
		}
	}
```

Asimismo, se deberán modificar las clases *Tree* y *Bush* para que extiendan la nueva clase *Plant*.

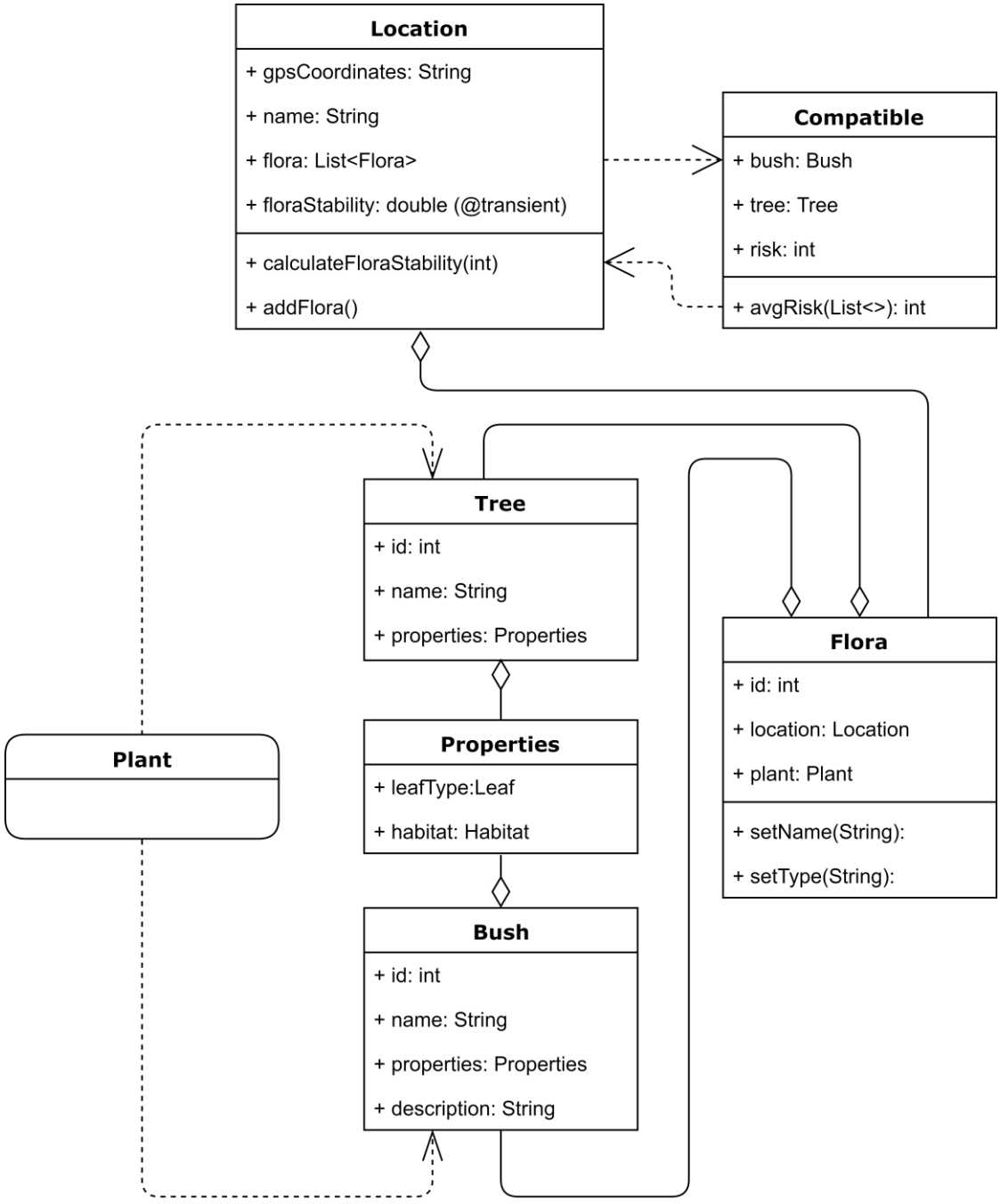
```
public abstract class Plant {

	public void setId(int id) {
	}

	public int getId() {
		return -1;
	}

	public String getName() {
		return null;
	}
}
```

```
@Entity
@Table(name = "TREES")
public class Tree extends Plant {

	. . .

	@Override
	public int getId() {
		return id;
	}

	@Override
	public void setId(int id) {
		this.id = id;
	}

	@Override
	public String getName() {
		return name;
	}

	. . .

}
```

```
@Entity
@Table(name = "BUSHES")
public class Bush extends Plant {

	. . .

	@Override
	public int getId() {
		return id;
	}

	@Override
	public void setId(int id) {
		this.id = id;
	}
```

```
        @Override
        public String getName() {
                return name;
        }

        . . .

}
```

El diagrama final del modelo de domino debería quedar como lo que sigue.



Para ello se añadirá el atributo *floraStability* como *@Transient* de la entidad *Location* y unos métodos simples para trabajar con él.

```
@Entity
@Table(name = "LOCATED")
public class Location {

	. . .

	@Transient
	private double floraStability;

	. . .

	public double getFloraStability() {
		return this.floraStability;
	}

	public void setFloraStability(int i) {
		this.floraStability = i;
	}

	public void addRisk(int risk) {
		this.floraStability = this.floraStability + risk;
	}

	public void averageStability(int num) {
		this.floraStability = this.floraStability/num;
	}

	. . .
 }
```

Y se creará la entidad *Compatible* para extraer los datos de riesgos de la base de datos.

```
@Entity
@Table(name = "COMPATIBLES")
public class Compatible implements Serializable {

	private static final long serialVersionUID = 1L;
	@Id
	@Column(name = "COMPATIBLE_ID")
	private int id;
	@Column(name = "TREE_ID")
	private int treeId;
	@Column(name = "BUSH_ID")
	private int bushId;
	@Column(name = "COMPATIBLE_RISK")
	private int risk;

	. . .

	@Override
	public int hashCode() {
		return super.hashCode();
	}
```

```
	@Override
	public boolean equals(Object obj) {
		return obj instanceof Compatible ?
		((Compatible) obj).getId() == this.treeId : false;
	}
}
```

Finalmente se creará un método *main()* para probar el correcto funcionamiento de las entidades.

Dado un objeto *Location*, se obtiene el atributo *floraStability* como la media ponderada de los *COMPATIBLES_RISK* de su atributo *flora*.

```
public class Program {

	public static void main(String[] args) {

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

	. . .

		Location l = Service.buscarLocation(em, 1);

		// Hay que buscar por nombre las Plants de flora
		for (Flora f : l.getFlora()) {
			if (f.getPlant() instanceof Tree) {
				f.setPlant(Service.buscarTree(em, f.getPlant().getId()));
			} else if (f.getPlant() instanceof Bush) {
				f.setPlant(Service.buscarBush(em, f.getPlant().getId()));
			}
		}
		// comprobamos que las plantas ya han sido cargadas
		System.out.println(Service.buscarFlora(em, 1) + "\nLocation 1: " +
		l.getFlora());
		System.out.println("-----------------");

		// ----- separamos las especies para calculo final de la estabilidad
		List<Tree> trees = new ArrayList<Tree>();
		List<Bush> bushes = new ArrayList<Bush>();
		for (Flora f : l.getFlora()) {
			if (f.resolvePlantType().equals("Tree")) {
				trees.add((Tree) f.getPlant());
			} else if (f.resolvePlantType().equals("Bush")) {
				bushes.add((Bush) f.getPlant());
			}
		}

		System.out.println("La lista de arboles es: " + trees + "\nLa lista de
		arbustos es: " + bushes);
		System.out.println("-----------------");

		// calculamos la estabilidad
		List<Compatible> compatible = Service.buscarCompatibles(em);
		l.setFloraStability(0);
		int relaciones = 0;
```

```
                for (Compatible c : compatible) {
                        for (Bush b : bushes) {
                                for (Tree t : trees) {
                                        if (c.getBushId() == b.getId() && c.getTreeId() ==
                                        t.getId()) {
                                                System.out.println("relación " + b.getName() +
                                                " con " + t.getName() + ", riesgo: " +
                                                c.getRisk());
                                                relaciones++;
                                                l.addRisk(c.getRisk());
                                        }
                                }
                        }
                }

                l.averageStability(relaciones);
                System.out.println("La estabilidad de '" + l.getName() + "' es de " +
                l.getFloraStability());

                em.close();
                emf.close();
        }
}
```

Para terminar, se han trabajado en mayor profundidad con las anotaciones para el *mixed Access*:

```
@Access(AccessType.FIELD)
```

```
@Access(AccessType.PROPERTY)
```

Y con la anotación:

```
@Transient
```

Concluyendo así las partes más relevantes del capítulo.

El objetivo final de este tutorial es comprender la potencia del ORM en el mapeo de entidades y relaciones ilustrando la gran diferencia conceptual que se puede alcanzar en el mapeado del modelo físico hacia las necesidades del modelo de domino.

3. Tutorial Capítulo 4

En este capítulo se desarrollará un análisis y uso del lenguaje JP QL (Java Persistance Query Language). Como base de datos se va a utilizar la dada por Oracle, *hr*.

3.1. Creación de tablas e inserción de datos

Se va a usar la base de datos *hr* de Oracle con tres tablas nuevas:

- Tabla **Director**, cuyos campos serán un *number()* **id_director** que lo identifique de manera única, un *varchar2* **nombre** con el nombre y apellidos del director y un *varchar2* **nacionalidad** con la nacionalidad del director.
- Tabla **Productora**, cuyos campos serán un *number()* **id_productora** que lo identifique de manera única, un *varchar2* **nombre** con el nombre de la productora y un *varchar2* **pais** con el país al que pertenezca la productora.
- Tabla **Película**, con campos *number()* **id_pelicula** que será el identificador de la película, un *varchar2* ***titulo***, un *number()* **año** con el año de lanzamiento, un *varchar2* **genero** y dos *number(),* **presupuesto** y **recaudacion** la recaudación y el presupuesto de la película. Además, se debe incluir el director y la productora encargados de la película dentro de la tabla, como claves foráneas.

```
CREATE TABLE Director(
	id_director number(10) primary key,
	nombre varchar2(100),
	nacionalidad varchar2(100)
);

CREATE TABLE Productora(
	id_productora number(10) primary key,
	nombre varchar2(100),
	pais varchar2(100)
);


CREATE TABLE Pelicula(
	id_pelicula number(10) primary key,
	titulo varchar2(100),
	año number(5),
	genero varchar2(100),
	director number(10),
	productora number(10),
	recaudacion number(10, 2),
	presupuesto number(10, 2)
);
```

```
ALTER TABLE pelicula
add constraint director foreign key(director) references Director(id_director);
ALTER TABLE pelicula
add constraint productora foreign key(productora) references Productora(id_productora);

INSERT INTO Director VALUES(1,'Steven Spielberg','Estadounidense');
INSERT INTO Director VALUES(2,'Patty Jenkins','Estadounidense');
INSERT INTO Director VALUES(3,'Julius Avery','Australiano');
INSERT INTO Director VALUES(4,'Guy Ritchie','Británico');
INSERT INTO Director VALUES(5,'Hermanas Wachowski','Estadounidense');
INSERT INTO director VALUES (6, 'Santiago Segura', 'Española');

INSERT INTO Productora VALUES(1,'Warner Bros','Estados Unidos');
INSERT INTO Productora VALUES(2,'Paramount Pictures','California');
INSERT INTO Productora VALUES(3,'20th Century Fox','Estados Unidos');

INSERT INTO Pelicula VALUES(1, 'Ready Player One: Comienza el juego', 2018, 'Ciencia Ficción', 1, 1, 582, 175);
INSERT INTO Pelicula VALUES(2, 'Mujer Maravilla', 2017, 'Ciencia Ficción', 2, 1, 822, 149);
INSERT INTO Pelicula VALUES(3, 'Overlord', 2018, 'Terror', 3, 2, 84.7, 38);
INSERT INTO Pelicula VALUES(4, 'Snatch, cerdos y diamantes', 2000, 'Acción', 4, 3, 93.6, 6);
INSERT INTO Pelicula VALUES(5, 'Pruebas Varias', 1982, 'Comedia', 2, 3, 35.2, 10);
INSERT INTO Pelicula VALUES(6, 'Otras Pruebas', 2018, 'Comedia', 4, 2, 94.5, 15);
INSERT INTO Pelicula VALUES (7, 'Torrente', 1998, 'Comedia', 6, null, 10.9, 12.7);
INSERT INTO Pelicula VALUES(8, 'Pruebas Null', 2018, 'Comedia', null, null, 100, 30);
INSERT INTO Pelicula VALUES(9, 'E.T', 1982, 'Ciencia Ficción', 1, 1, 730, 10.5);
INSERT INTO Pelicula VALUES(10, 'Matrix', 1999, 'Ciencia Ficción', 5, 1, 464.5, 63);
INSERT INTO Pelicula VALUES(11, 'Jurassic Park', 1993, 'Ciencia Ficción', 1, 1, 1029, 63);
```

3.2. Entidades

A continuación, como en el tema 2, el tutorial necesitará la creación de entidades que puedan ser el reflejo en Java de las tablas creadas en la base de datos. A continuación, el código de las entidades:

```
@Entity
@Table(name="DIRECTOR")
public class Director implements Serializable {

	private static final long serialVersionUID = 1L;
	@Id
	@Column(name="ID_DIRECTOR")
	private int id_director;
	@Column(name="NOMBRE")
	private String nombre;
	@Column(name="NACIONALIDAD")
	private String nacionalidad;

	@OneToMany(mappedBy = "director", cascade = CascadeType.PERSIST)
	private List<Pelicula> peliculas = new ArrayList<>();

	public Director() {

	}

	public Director(int id, String nombre, String nacionalidad) {
		super();
		this.id_director = id;
		this.nombre = nombre;
		this.nacionalidad = nacionalidad;
	}
	.
	.
	[Introducir aquí métodos getters y setters]
	.
	.
	@Override
	public boolean equals(Object elOtro) {
		if (elOtro instanceof Director) {
			Director d = (Director) elOtro;
			return this.id_director == d.id_director;
		}
		return false;
	}

	@Override
	public int hashCode() {
		return (int) this.id_director;
	}

	@Override
	public String toString() {
	  return String.format("id=%d, nombre=%s, nacionalidad=%s", this.id_director,
			this.nombre, this.nacionalidad);
	}

	//Método para poder imprimir por pantalla solo el nombre del director
	public String toStringPelicula() {
		return String.format("%s", this.nombre);
	}

}
```

```
@Entity
@Table(name="PRODUCTORA")
public class Productora implements Serializable {

	private static final long serialVersionUID = 1L;
	@Id
	@Column(name="ID_PRODUCTORA")
	private int id_productora;
	@Column(name="NOMBRE")
	private String nombre;
	@Column(name="PAIS")
	private String pais;


	@OneToMany(mappedBy = "productora", cascade = CascadeType.PERSIST)
	private List<Pelicula> peliculas = new ArrayList<>();

	public Productora() {

	}

	public Productora(int id, String nombre, String pais){
		super();
		this.id_productora = id;
		this.nombre = nombre;
		this.pais = pais;
	}
	.
	.
	.
	[Introducir aquí métodos getters y setters]
	.
	.
	.
	@Override
	public boolean equals(Object laOtra) {
		if (laOtra instanceof Productora) {
			Productora p = (Productora) laOtra;
			return this.id_productora == p.id_productora;
		}
		return false;
	}

	@Override
	public int hashCode() {
		return (int) this.id_productora;
	}

	@Override
	public String toString() {
	  return String.format("id=%d, nombre=%s, pais=%s", this.id_productora, this.nombre,
			this.pais);
	}

	public String toStringPelicula() {
		return String.format("%s",this.nombre);
	}

}
```

```
@Entity
```

```java
@Table(name="PELICULA")
public class Pelicula implements Serializable {

	private static final long serialVersionUID = 1L;
	@Id
	@Column(name="ID_PELICULA")
	private int id_pelicula;
	@Column(name="TITULO")
	private String titulo;
	@Column(name="AÑO")
	private int año;
	@Column(name="GENERO")
	private String genero;
	@Column(name="RECAUDACION")
	private double recaudacion;
	@Column(name="PRESUPUESTO")
	private double presupuesto;
	@ManyToOne
	@JoinColumn(name="DIRECTOR")
	private Director director;
	@ManyToOne
	@JoinColumn(name="PRODUCTORA")
	private Productora productora;


	public Pelicula() {

	}

	public Pelicula(int id_pelicula, String titulo, int año, String genero, double
	recaudacion, double presupuesto, Director director, Productora productora) {
		super();
		this.id_pelicula = id_pelicula;
		this.titulo = titulo;
		this.año = año;
		this.genero = genero;
		this.recaudacion = recaudacion;
		this.presupuesto = presupuesto;
		this.director = director;
		this.productora = productora;
	}
	.
	.
	.
	[Introducir aquí métodos getters y setters]
	.
	.
	.


	@Override
	public boolean equals(Object elOtro) {
		if (elOtro instanceof Pelicula) {
			Pelicula l = (Pelicula) elOtro;
			return this.id_pelicula == l.id_pelicula;
		}
		return false;
	}
```

Para poder imprimir por pantalla el resultado de una consulta que una la entidad Pelicula con las otras dos, pudiendo ser el valor de cualquiera de ellas *null* (sin director o sin productora), en el método *toString* se han dado las diferentes opciones.

```
@Override
public String toString() {
      if(this.director==null && this.productora==null)
        return String.format("id=%s, titulo=%s, año=%d, genero=%s, recaudacion=%.2f,
               presupuesto=%.2f, director =null, productora=null,", this.id_pelicula,
               this.titulo, this.año, this.genero, this.recaudacion, this.presupuesto);

      else if(this.productora==null && this.director != null)
        return String.format("id=%s, titulo=%s, año=%d, genero=%s, recaudacion=%.2f,
               presupuesto=%.2f, director =%s, productora=null,", this.id_pelicula,
               this.titulo, this.año, this.genero, this.recaudacion, this.presupuesto,
               this.director.toStringPelicula());

      else if(this.director==null && this.productora != null)
        return String.format("id=%s, titulo=%s, año=%d, genero=%s, recaudacion=%.2f,
               presupuesto=%.2f, director =null, productora=%s,", this.id_pelicula,
               this.titulo, this.año, this.genero, this.recaudacion, this.presupuesto,
               this.productora.toStringPelicula());

      else
        return String.format("id=%s, titulo=%s, año=%d, genero=%s, recaudacion=%.2f,
               presupuesto=%.2f, director =%s, productora=%s,", this.id_pelicula,
               this.titulo, this.año, this.genero, this.recaudacion, this.presupuesto,
               this.director.toStringPelicula(),this.productora.toStringPelicula());
        }

}
```

Crear a continuación las clases Servicio, una por cada entidad (ServicioDirector, ServicioProductora, ServicioPelicula). También crear la clase Programa para ejecutar el método *main()*.

Todo esto viene explicado en el tema 2 y puede ver el código utilizado para este capítulo en el TutorialJPA.zip.

3.3. Queries

El tutorial emplea sobre todo `@NamedQueries`, debido a su facilidad de ordenación y utilización más clara. A pesar de ello, `@TypedQueries` nos da las mismas posibilidades de consulta. Las *queries* que nombremos se escribirán en el código de la entidad y las `TypedQueries` directamente en la clase Servicio de esa entidad. La anotación `@NamedQueries` y sus `@NamedQuery` correspondientes se escriben debajo de la anotación `@Entity`.

3.3.1. Consultas SELECT

```
@NamedQueries({
@NamedQuery(name = "pelicula.all", query = "SELECT p FROM Pelicula p "),
.
.
[Aquí se introducirá cualquier @NamedQuery que se desee y tenga como objetivo de la consulta, la entidad Pelicula o alguno de sus campos]

})
```

En la clase ServicioPelicula se creará el método que utilice la `@NamedQuery` anterior:

```
public List<Pelicula> getPeliculasAll(EntityManager em) {
 List<Pelicula> peliculas = new ArrayList<Pelicula>();
 peliculas = em.createNamedQuery("pelicula.all", Pelicula.class).getResultList();
return peliculas;
}
```

Este método devolverá una lista de todas las películas que se encuentren en base de datos. Para el programa principal se utiliza uno similar al capítulo 2, utilizando `EntityManager()`, y `EntityManagerFactory()`. El código de la clase Programa será el siguiente para la mayoría de las consultas a realizar:

```
public class Programa {
 public static void main(String[] args){
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("cine");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        List<Pelicula> l = ServicioPelicula.getPeliculasAll(em);
        for(Pelicula p:l)
        System.out.println(p.toString());

        em.getTransaction().commit();
        em.close();
        emf.close();
 }

}
```

El resultado de esta consulta será:

id=1, titulo=Ready Player One: Comienza el juego, año=2018, genero=Ciencia Ficción, recaudacion=582,00, presupuesto=175,00, director =Steven Spielberg, productora=Warner Bros,

id=2, titulo=Mujer Maravilla, año=2017, genero=Ciencia Ficción, recaudacion=822,00, presupuesto=149,00, director =Patty Jenkins, productora=Warner Bros,

id=3, titulo=Overlord, año=2018, genero=Terror, recaudacion=84,70, presupuesto=38,00, director =Julius Avery, productora=Paramount Pictures,

id=4, titulo=Snatch, cerdos y diamantes, año=2000, genero=Acción, recaudacion=93,60, presupuesto=6,00, director =Guy Ritchie, productora=20th Century Fox,

id=5, titulo=Pruebas Varias, año=1982, genero=Comedia, recaudacion=35,20, presupuesto=10,00, director =Patty Jenkins, productora=20th Century Fox,

id=6, titulo=La vida es Bella, año=2018, genero=Comedia, recaudacion=94,50, presupuesto=15,00, director =Guy Ritchie, productora=Paramount Pictures,

id=7, titulo=Torrente, año=1998, genero=Comedia, recaudacion=10,90, presupuesto=12,70, director =Santiago Segura, productora=null,

id=8, titulo=Pruebas Null, año=2018, genero=Comedia, recaudacion=100,00, presupuesto=30,00, director =null, productora=null,

id=9, titulo=E.T, año=1982, genero=Ciencia Ficción, recaudacion=730,00, presupuesto=10,50, director =Steven Spielberg, productora=Warner Bros,

id=10, titulo=Matrix, año=1999, genero=Ciencia Ficción, recaudacion=464,50, presupuesto=63,00, director =Hermanas Wachowski, productora=Warner Bros,

id=11, titulo=Jurassic Park, año=1993, genero=Ciencia Ficción, recaudacion=1029,00, presupuesto=63,00, director =Steven Spielberg, productora=Warner Bros,

La consulta anterior también puede ser realizada mediante una `@TypedQuery` de la siguiente manera:

```
public static Pelicula TPQuery(EntityManager em){
		TypedQuery<Pelicula> p = em.createQuery(
		"SELECT p FROM Pelicula p WHERE p.id_pelicula = 4", Pelicula.class);
		try {
			return p.getSingleResult();
			}
		catch (NoResultException e) {
			return null;
			}

}
```

3.3.2. Consultas WHERE

```
@NamedQuery(name = "pelicula.por.genero", query = "SELECT p FROM Pelicula p WHERE p.genero =:genero")
```

En la clase ServicioPelicula:

```
public static List<Pelicula> getPeliculasGenero(EntityManager em) {
        String a = "Ciencia Ficción";
        List<Pelicula> peliculas = new ArrayList<Pelicula>();
        peliculas = em.createNamedQuery("pelicula.por.genero",
            Pelicula.class).setParameter("genero",a ).getResultList();
 return peliculas;

}
```

En el método anterior se crea una variable de tipo *String*, la cual corresponde con un género de película. Aplicando el método *`setParameter()`* a la consulta podemos realizar consultas dinámicas.

El resultado de la consulta anterior es:

```
id=1, titulo=Ready Player One: Comienza el juego, año=2018, genero=Ciencia Ficción, recaudacion=582,00, presupuesto=175,00, director =Steven Spielberg, productora=Warner Bros,

id=2, titulo=Mujer Maravilla, año=2017, genero=Ciencia Ficción, recaudacion=822,00, presupuesto=149,00, director =Patty Jenkins, productora=Warner Bros,

id=9, titulo=E.T, año=1982, genero=Ciencia Ficción, recaudacion=730,00, presupuesto=10,50, director =Steven Spielberg, productora=Warner Bros,

id=10, titulo=Matrix, año=1999, genero=Ciencia Ficción, recaudacion=464,50, presupuesto=63,00, director =Hermanas Wachowski, productora=Warner Bros,

id=11, titulo=Jurassic Park, año=1993, genero=Ciencia Ficción, recaudacion=1029,00, presupuesto=63,00, director =Steven Spielberg, productora=Warner Bros,
```

Otro tipo de consulta empleando expresiones de ruta:

```
@NamedQuery(name = "pelicula.por.director", query = "SELECT p FROM Pelicula p WHERE p.director.nombre =:director")
```

```
public static List<Pelicula> getPeliculasDirector(EntityManager em) {
 String a = "Guy Ritchie";
 List<Pelicula> peliculas = new ArrayList<Pelicula>();
 peliculas = em.createNamedQuery("pelicula.por.director",
       Pelicula.class).setParameter("director",a ).getResultList();

return peliculas;
```

Resultado de la consulta:

```
id=4, titulo=Snatch, cerdos y diamantes, año=2000, genero=Acción, recaudacion=93,60, presupuesto=6,00, director =Guy Ritchie, productora=20th Century Fox,

id=6, titulo=La vida es Bella, año=2018, genero=Comedia, recaudacion=94,50, presupuesto=15,00, director =Guy Ritchie, productora=Paramount Pictures,
```

Distintas consultas con WHERE

Esta consulta devolverá las películas que hayan sido estrenadas a partir del año 2001.

```
@NamedQuery(name = "pelicula.siglo21", query = "SELECT p FROM Pelicula p WHERE p.año > 2000"),
```

En esta ocasión se utiliza BETWEEN para consultar qué películas han generado entre 500 y 1000 millones de euros.

```
@NamedQuery(name = "pelicula.between", query = "SELECT p FROM Pelicula p WHERE p.recaudacion BETWEEN 500 AND 1000"),
```

La consulta siguiente devolverá las películas pertenecientes a un género cuyo nombre termine con la cadena de texto 'cció' seguida por cualquier carácter.

```
@NamedQuery(name = "pelicula.genero", query = "SELECT p FROM Pelicula p WHERE p.genero LIKE '%cció_' "),
```

Esta consulta devolverá la película con mayor recaudación utilizando una subconsulta y la palabra agregada MAX.

```
@NamedQuery(name = "pelicula.maxrecaudacion", query = "SELECT p FROM Pelicula p WHERE p.recaudacion = (SELECT MAX(pel.recaudacion) FROM Pelicula pel)"),
```

La consulta siguiente devolverá las películas, donde el director de ellas no sea estadounidense.

```
@NamedQuery(name = "pelicula.NotInUSA", query = "SELECT p FROM Pelicula p WHERE p.director.nacionalidad NOT IN ('Estadounidense') "),
```

Esta consulta devolverá las películas que no tengan productora.

```
@NamedQuery(name = "pelicula.SinProductora", query = "SELECT p FROM Pelicula p WHERE p.productora IS NULL"),
```

La siguiente consulta devolverá las películas cuya productora no sea 'Warner Bros'

```
@NamedQuery(name = "pelicula.NotWB", query = "SELECT p FROM Pelicula p WHERE NOT EXISTS (SELECT prod FROM p.productora prod WHERE prod.nombre = 'Warner Bros')"),
```

3.3.3. Consultas JOIN

```
@NamedQuery(name = "pelicula.join.director.where", query = "SELECT p.titulo,d.nombre
FROM Pelicula p JOIN p.director d WHERE d.nombre ='Steven Spielberg' "),
```

La consulta anterior une las tablas Pelicula y Director. En este caso devolvemos el título de la película y el nombre del director, es decir, campos de entidades distintas. Por ello el método empleado para devolver la lista cambia ligeramente. A partir de ahora, siempre que se unan tablas, se tendrá que devolver una lista de objetos, no de películas, como se estaba hacienod anteriormente:

Clase ServicioPelicula :

```
public static List<Object[]> getPeliculasJoinDirectorWhere(EntityManager em) {
 List<Object[]> peliculas = new ArrayList<Object[]>();
 peliculas = em.createNamedQuery("pelicula.join.director.where",
     Object[].class).getResultList();
return peliculas;
}
```

La consulta deberá devolver todas las películas cuyo director tenga el nombre de 'Steven Spielberg':

```
[Ready Player One: Comienza el juego, Steven Spielberg]
[E.T, Steven Spielberg]
[Jurassic Park, Steven Spielberg]
```

Hay que tener en cuenta que, para obtener estos resultados, será necesario modificar, en el método *main*, la manera en la que se imprimen los mismos debido a que el método de ServicioPelicula ahora devuelve una lista de objetos y no una lista de películas. Así, se tendría en *main* las siguientes líneas:

```
List<Object[]> l = ServicioPelicula.getPeliculasJoinDirectorWhere(em);
for(Object[] p:l){
 System.out.println(Arrays.toString(p));
}
```

La siguiente consulta, une las tablas Pelicula y Productora, pero en este caso devolverá todas las películas tengan o no productora. El método empleado en la clase ServicioPelicula es prácticamente idéntico al anterior.

```
@NamedQuery(name = "pelicula.leftjoin.productora", query = "SELECT p.titulo,d.nombre
FROM Pelicula p LEFT JOIN p.productora d"),
```

```
[Jurassic Park, Warner Bros]
[Matrix, Warner Bros]
[E.T, Warner Bros]
[Mujer Maravilla, Warner Bros]
[Ready Player One: Comienza el juego, Warner Bros]
[La vida es Bella, Paramount Pictures]
[Overlord, Paramount Pictures]
[Pruebas Varias, 20th Century Fox]
[Snatch, cerdos y diamantes, 20th Century Fox]
[Pruebas Null, null]
[Torrente, null]
```

3.3.4. Consultas Group By

```
@NamedQuery(name = "pelicula.groupBy.having", query = "SELECT d.nombre, COUNT(p) FROM
Pelicula p LEFT JOIN p.productora d GROUP BY(d.nombre) HAVING COUNT(p)>1")
```

Esta consulta devolverá el número de películas que ha realizado cada productora, agrupándolas por nombre y eliminando aquellas que no hayan hecho más de 1.

Clase ServicioPelicula:

```
public static List<Object[]> getPeliculasGroupByHaving(EntityManager em) {
        List<Object[]> peliculas = new ArrayList<Object[]>();
        peliculas = em.createNamedQuery("pelicula.groupBy.having",
        Object[].class).getResultList();
 return peliculas;

}
```

Como se puede observar, cualquier consulta en la que se quiera devolver más de un valor se debe emplear un método que devuelva una lista de *arrays* de *Object*.

```
[20th Century Fox, 2]
[null, 2]
[Warner Bros, 5]
[Paramount Pictures, 2]
```

3.3.5. Consultas varias

Otras palabras reservadas permiten poner condiciones distintas a las consultas y poder rescatar aquellos datos que queramos. Algunos ejemplos son los siguientes:

- En esta consulta se seleccionan aquellas películas cuya recaudación es menor que cualquier de los presupuestos de todas las películas.

```
@NamedQuery(name = "pelicula.ANY", query = "SELECT p FROM Pelicula p WHERE
p.recaudacion < ANY(SELECT pelic.presupuesto FROM Pelicula pelic) ORDER BY
p.presupuesto DESC"),
```

- En este caso se utiliza la palabra CASE para cambiar el género de la película Matrix y Jurassic Park. El método de ServicioPelicula encargado de devolver los resultados de esta consulta devolverá una lista de objetos.

```
@NamedQuery(name = "pelicula.CASE", query = "SELECT p.titulo, "
 + "CASE p.titulo WHEN 'Matrix' THEN 'Filosofica' "
 + "WHEN 'Jurassic Park' THEN 'Dinosaurios' "
 + "ELSE p.genero "
 + "END FROM Pelicula p"),
```

- En esta consulta se recuperará la media de todos los presupuestos y de las recaudaciones. El método de ServicioPelicula encargado de devolver los resultados de esta consulta también devolverá una lista de objetos.

```
@NamedQuery(name = "pelicula.AVG", query = "SELECT AVG(p.presupuesto),
AVG(p.recaudacion) FROM Pelicula p"),
```

3.3.6. Consulta UPDATE

Para modificar cualquier registro en la base de datos, utilizamos la sentencia UPDATE, donde podemos cambiar el valor de cualquier de las propiedades de una entidad.

```
@NamedQuery(name = "pelicula.UPD", query = "UPDATE Pelicula p SET p.titulo = 'La vida
es Bella' WHERE p.id_pelicula = 6 "),
```

El método para la clase ServicioPelicula, difiere de los anteriores siendo del modo:

```
public static void updatePelicula(EntityManager em) {
                    em.createNamedQuery("pelicula.UPD").executeUpdate();

}
```

3.3.7. Consulta DELETE

Para borrar cualquier registro en la base de datos, utilizamos la sentencia DELETE.

```
@NamedQuery(name = "pelicula.DEL", query = "DELETE FROM Pelicula p WHERE p.id_pelicula
= 12"),
```

Este método (de la clase ServicioPelicula), será el mismo que para UPDATE, teniendo en cuenta el cambio de query.

```
public static void deletePelicula(EntityManager em) {
                    em.createNamedQuery("pelicula.DEL").executeUpdate();
}
```