Implementación del nivel de datos usando ObjectDB

Introducción

En este laboratorio, instalaremos y utilizaremos las clases necesarias para acceder a bases de datos orientadas a objetos ObjectDB. ObjectDB es un software comercial, pero que puede usarse de manera gratuita siempre que las BDs no tengan más de 10 clases entidad ni más de un millón de objetos. Aunque ObjectDB sea un software propietario, las APIs en las que se basa no; son APIs estándar de Java, como por ejemplo Java Persistence API (JPA), que es la que usaremos.

Objetivos

En este laboratorio se pretende que se sepa realizar lo siguiente:

- Instalar el SGBD objectDB
- Abrir y cerrar bases de datos objectDB
- Realizar operaciones básicas CRUD con bases de datos objectDB (Create=crear, Read=leer, Update=actualizar, Delete=borrar)
- Realizar consultas a BDs objectDB
- Implementar la lógica del negocio accediendo al nivel de datos

Pasos a seguir

1. Instalación de objectDB y creación de proyecto para objectDB

Los pasos para instalar la versión 2.X de objectDB y crear un proyecto Eclipse que trabaje con bases de datos orientadas o objetos ObjectDB son los siguientes:

a) Descargar la instalación de ObjectDB desde su web http://www.objectdb.com, en el apartado de Download.

b) Descomprimir el fichero. Entre su contenido se encuentra el fichero .jar que contiene el gestor de base de datos (**objectdb-2.X_**/bin/objectdb.jar**) y la herramienta **ObjectDb Explorer** que permite visualizar los objetos de las bases de datos objectDB (**objectdb-2.X_**/bin/explorer.exe)**

c) Crear un proyecto Eclipse (por ejemplo **ObjectdbLab-1**) con una carpeta llamada **lib**

Nota: aunque uséis un JRE posterior a 1.8, escoged la versión 1.8 de Java

d) Copiar el fichero objectdb.jar a la carpeta lib

e) Añadir el fichero .jar al proyecto

Project => Properties => Java Build Path => Libraries => Add JAR => Seleccionar proyecto ObjectdbLab-1 => seleccionar lib => objectdb.jar => OK

2. Crear una clase del dominio (a la que se quiere dar persistencia)

Definir la clase Pilot en el paquete domain (es una clase del modelo del dominio) cuyos objetos deseamos guardar de manera persistente:

```
package domain;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Pilot {
  @Id
	String name;
	String nationality;
	int points;

	public Pilot(String name, String naz, int x){
		this.name=name;
		this.nationality =naz;
		this.points=x;
	 }
	public String  toString(){
		return name+" "+Integer.toString(points);
	}
	public void addPoints(int x){
		this.points= this.points+x;
	}
	public String getName(){
		return this.name;
	}
}
```

Tened en cuenta que es una clase normal Java (o clase POJO = Plain Old Java Object), que contiene las siguientes dos anotaciones Java:

a) La anotación @Entity que se añade a la clase (antes de **public class**)
b) Y la anotación @Id que se le añade a la propiedad name, y que la establece como clave principal, lo que significa que son objetos diferentes aquellos que tengan un valor distinto para name, y que dos objetos con el mismo valor para name son el mismo. Esto quiere decir que los pilotos se diferenciarán por el nombre: no será posible guardar en la base de datos objectdb dos pilotos con el mismo nombre.(aunque en Java sí podría ser posible tener dos objetos diferentes con el mismo nombre, como ya es conocido).

Por el hecho de que se define un objeto Java (por ejemplo con: `new Pilot("Vettel","Germany",25))` no se obtiene automáticamente la persistencia del mismo. De momento, en la clase Java Pilot no se ha incluido código que asegure la persistencia: la clase está preparada para que se pueda dar persistencia a los objetos Java.

3. Crear una clase Java para abrir y cerrar la BD.

Crear una clase Java (por ejemplo F1_objectdbAccess) y definir las llamadas para abrir y cerrar una base de datos, tal y como aparece en la siguiente clase:

```
package dataAccess;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class F1_objectdbAccess {
	private  EntityManager  db;
	private EntityManagerFactory emf;
	String fileName = "formula1.odb";

	public F1_objectdbAccess() {
	  emf = Persistence.createEntityManagerFactory("objectdb:"+fileName);
	   db = emf.createEntityManager();
	   System.out.println("Base de datos abierta");
	   }

public void close(){
	   db.close();
	   System.out.println("Base de datos cerrada");
	}
}
```

En este caso, la base de datos objectDB se abre en el constructor de la clase y se almacena en un fichero con extensión odb (en este caso en el fichero `formula1.odb`) que se guardará en el directorio raíz donde se encuentre el proyecto Eclipse. Tras abrir la base de datos, en el atributo db de la clase se guarda una referencia de la base datos (de tipo **EntityManager**) que permitirá trabajar con la misma. Si no existe la base de datos, se crea un nuevo fichero.

4. Insertar objetos en la base de datos

En la clase F1_objectdbAccess definiremos el siguiente método que permite insertar nuevos objetos en la BD:

```
 public void storePilot(String name,String nac, int points) {

	db.getTransaction().begin();
	Pilot pilot = new Pilot(name, nac, points);
	db.persist(pilot);
	db.getTransaction().commit();

	System.out.println("Insertado: " + pilot);
}
```

Todas las operaciones (en particular las que modifican el contenido de la base de datos) hay que ejecutarlas dentro de transacciones. Una transacción está formada por varias operaciones con la base de datos que se desea se ejecuten como una unidad: o se ejecutan todas o no se ejecuta ninguna. En este caso una transacción empieza

invocando a db.getTransaction().begin() y termina con db.getTransaction().commit(). Para crear objetos se utiliza el método **persist**.

Nota: la llamada al método commit (db.getTransaction().commit()) indica de manera explícita que los cambios realizados en la BD deben considerarse como definitivos. Si no lo hiciéramos, entonces al cerrar la BD de manera normal (`db.close()`), entonces implícitamente también se haría.

Para probar el método anterior crearemos una nueva clase (Races) que acceda al nivel de datos (y que por tanto consideraremos como parte de la lógica del negocio, por lo que la asignaremos al paquete bussiness):

```
package bussiness;
import dataAccess.F1_objectdbAccess;

public class Races {
	public static void main(String[] args) {
		F1_objectdbAccess f1_dataMng=new F1_objectdbAccess();
		f1_dataMng.storePilot("Jesus Magarri", "Mexican", 405);
		f1_dataMng.storePilot("Sebastian Vettel", "German", 345);
		f1_dataMng.close();}
}
```

Insertad 6 nuevos pilotos en la BD de entre los que se encuentran en la siguiente URL (uno de ellos que sea Fernando Alonso):

http://www.formula1.com/teams_and_drivers/drivers/

Comprobad que al ejecutar la clase Races aparecen los mensajes apropiados y que se ha creado el fichero **formula1.odb** en el directorio raíz del proyecto.

5. Realizar consultas a la base de datos

5.1. Consultar un único objeto conocido el valor de la clave (Id)

Si se conoce el valor exacto del atributo clave, entonces se puede recuperar el objeto invocando al método find:

```
db.find(Pilot.class, "Jesus Magarri");
```

Se puede definir el siguiente método en la clase F1_objectdbAccess:

```
public Pilot getPilotById(String name) {
	return db.find(Pilot.class,name);
	}
```

Y probarlo añadiendo y ejecutando la siguiente instrucción en Races:

```
		System.out.println(f1_dataMng.getPilotById("Jesus Magarri"));
```

NOTA: en este punto **saltará un error** porque se está intentando insertar objetos que existen en la BD (con `f1_dataMng.storePilot("Jesus Magarri", "Mexican", 405);`)

Para evitarlo se puede hacer una de las siguientes opciones:

1) Comentar las instrucciones que crean los pilotos (puesto que ya se han ejecutado)

```
// f1_dataMng.storePilot("Jesus Magarri", "Mexican", 405);
// f1_dataMng.storePilot("Sebastian Vettel", "German", 345);
```

2) Borrar la base de datos antes de crearla, por ejemplo añadiendo la instrucción **new** File(fileName).delete(); en el constructor de `F1_objectdbAccess`

```
public F1_objectdbAccess() {
 new File(fileName).delete();
 new File(filename+"$").delete();
 emf = Persistence.createEntityManagerFactory("objectdb:"+fileName);
 db = emf.createEntityManager();
 System.out.println("Base de datos abierta");
}
```

Borrar una base de datos sólo tiene sentido hacerlo antes de crearla de nuevo, pero no antes de abrirla para seguir trabajando con ella. Por lo tanto, no es recomendable hacerlo en este método `F1_objectdbAccess`, que sirve tanto para crear como para abrir la BD. Es mejor usar la opción 1) en este caso.

5.2. Consultas más generales usando JPQL

Para realizar consultas con la base de datos se utiliza **JPQL (JPA Query Language),** un lenguaje de preguntas orientado a objetos parecido a SQL.

Por ejemplo, para obtener todos los pilotos hay que definir la siguiente consulta:

```
TypedQuery<Pilot> query = db.createQuery("SELECT p FROM Pilot p", Pilot.class);
```

Que se envía a la BD de esta manera:

```
List<Pilot> pilots = query.getResultList();
```

Y obtiene la respuesta en una lista, esto es, en un objeto de tipo **java.util.List**, que es una colección Java (**Java Collection**).

Para probarlo, definiremos el siguiente método en la clase F1_objectdbAccess:

```
public void getAllPilots() {
        TypedQuery<Pilot> query = db.createQuery("SELECT p FROM Pilot p",Pilot.class);
        List<Pilot> pilots = query.getResultList();
        System.out.println("Contenido de la base de datos:");
         for (Pilot p:pilots) System.out.println(p.toString());
}
```

De nuevo se puede comprobar si los objetos guardados anteriormente están en la base de datos o no.

Y añadir la llamada al método de consulta getAllPilots en el main de Races:

```
f1_dataMng.getAllPilots();
```

Ejecutad la aplicación y comprobad que devuelve la respuesta correcta.

La consulta **SELECT p FROM Pilot p WHERE p.nationality='Mexican'** recupera los pilotos de nacionalidad mejicana. Se puede definir el método getPilotByNationality en la clase F1_objectdbAccess que permita consultar pilotos por su nacionalidad

```
public void getPilotByNationality(String nac) {
  TypedQuery<Pilot> query =
          db.createQuery("SELECT p FROM Pilot p WHERE p.nationality='"+nac+"'",Pilot.class);
 List<Pilot> pilots = query.getResultList();
 System.out.println("Contenido de la base de datos");
 for (Pilot p:pilots) System.out.println(p.toString());
}
```

Y probarlo añadiendo en Races: `f1_dataMng.getPilotByNationality("Mexican");`

Consejo para realizar consultas con parámetros ("Mexican" en el caso anterior):

La consulta anterior se puede definir utilizando un método más efectivo (y que además es más recomendable contra la inyección de código https://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL), que es utilizando preguntas parametrizadas. Además evita problemas de formatear adecuadamente las fechas cuando se envían como parámetros.

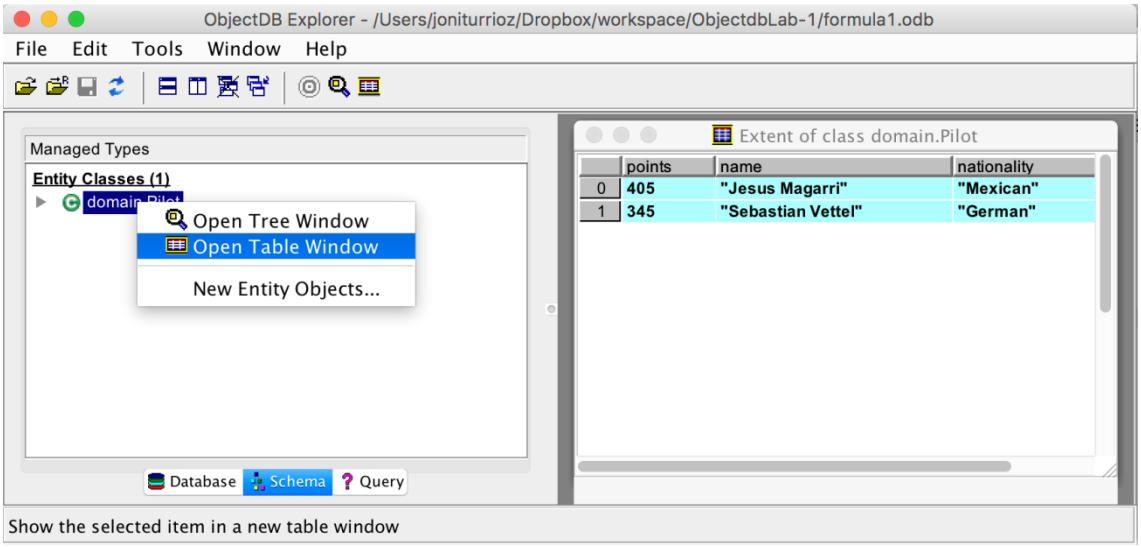
```
TypedQuery<Pilot> query =
    db.createQuery("SELECT p FROM Pilot p WHERE p.nationality=?1",Pilot.class);
query.setParameter(1, "Mexican");
```

De esta manera, se define una consulta con un parámetro (?1), al que más tarde se le puede asignar el valor concreto (Mexican, en este caso). Esto permitiría definir una única vez la consulta que podría ejecutarse después con distintos valores para el parámetro "nationality".

Ejercicio: cread un método que obtenga los pilotos por nacionalidad y utilice la pregunta parametrizada anterior y comprobad que el resultado es el mismo.

6. Explorar la base de datos

Con la instalación de objectdb, se descarga una aplicación que permite explorar los objetos de la base de datos. Dicha aplicación se ejecuta con el fichero **objectdb-2.X_**/bin/explorer.exe**. Por defecto abre la última base datos .odb que se ha modificado.



En el panel izquierdo aparecen las clases de la base de datos (el equivalente a tablas de bases de datos). Si nos situamos en la clase correspondiente y con el botón derecho seleccionamos “Open Table Window”, en el panel de la derecha aparecerán los objetos de dicha clase guardados en la base de datos.

Hay que acordarse de cerrar la aplicación, para que se pueda acceder a la base de datos desde la aplicación java y no dé un error de recurso abierto (y lo mismo sucede al revés).

7. Borrar objetos

7.1. Cuando el objeto está en memoria principal

Cuando el objeto que se desea borrar está en la base de datos y ha sido recuperado previamente con find o con una consulta JPQL, entonces se puede borrar con remove:

Se puede definir el siguiente método en la clase F1_objectdbAccess:

```
public void deletePilot(Pilot p) {
	db.getTransaction().begin();
	db.remove(p);
	db.getTransaction().commit();
}
```

Y se puede probar en Races con estas instrucciones:

```
Pilot jm=f1_dataMng.getPilotById("Jesus Magarri");
f1_dataMng.deletePilot(jm);
```

7.2. Borrado usando JPQL

Otra manera de borrar objetos es utilizando la operación DELETE de JPQL. Por ejemplo, para borrar el piloto Jesús Magarri se puede definir la siguiente instrucción que crea la operación de borrado

```
Query query = db.createQuery("DELETE FROM Pilot p WHERE p.name='"+name+"'");
```

Y a continuación la instrucción que ejecuta en la base de datos la operación de borrado, dejando en deletedPilots el número de objetos borrados en la base de datos:

```
int deletedPilots = query.executeUpdate();
```

El siguiente método, que se puede añadir a la clase F1_objectdbAccess, borra un piloto dado un nombre (name).

```
public void deletePilotByName(String name) {
 db.getTransaction().begin();
 Query query = db.createQuery("DELETE FROM Pilot p WHERE p.name='"+name+"'");
 int deletedPilots = query.executeUpdate();
 System.out.println("Pilotos borrados: " + deletedPilots);
 db.getTransaction().commit(); }
```

Ejercicio: añadir el código que borra al piloto **Fernando Alonso** y que a continuación muestra todos los pilotos para comprobar que ha funcionado correctamente.

8. Actualización objetos

Se puede realizar la actualización de objetos de una manera **transparente**. Para ello, si se cambia el valor de algún atributo de un objeto, al hacer el "commit" de la transacción, de manera automática y transparente se actualiza dicho objeto.

```
public void updatePilotByNameAddingPoints(String name, int points) {
       Pilot pilot=getPilotById(name);
       if (pilot==null)
           System.out.println(name + " no está en la base de datos");
          else {
                     db.getTransaction().begin();
                     pilot.addPoints(points);
                     //db.persist(pilot);
                     db.getTransaction().commit();
                  System.out.println(pilot + " ha sido actualizado");
             }
}
```

Esto es, no es necesario ejecutar la instrucción `db.persist(pilot)` que aparece comentada. Pero si se ejecutara el resultado sería el mismo.

Ejercicio: añadir el código que incremente 100 puntos al piloto **Sebastian Vettel** y que a continuación muestre todos los pilotos para comprobar que ha funcionado correctamente.

NOTA IMPORTANTE: Si al método para actualizar se le pasa como parámetro un objeto `p` de tipo `Pilot` en vez del nombre `name` del piloto, entonces **es necesario buscarlo con `find`** en la BD para asegurar que la actualización se hace exactamente en el objeto de la BD. Si se hubiera perdido la conexión entre el objeto en memoria principal y el objeto en la BD (cosa que ocurre cuando se invoca desde la máquina de presentación a la máquina con la lógica del negocio) entonces intentaría crear un nuevo objeto en la BD.

```
public void updatePilotAddingPoints(Pilot p, int points) {
      Pilot pilot= db.find(Pilot.class,p.getName());
      if (pilot==null)
          System.out.println(p.getName() + " no está en la base de datos");
         else {
                    db.getTransaction().begin();
                    pilot.addPoints(points);
                    db.getTransaction().commit();
                 System.out.println(pilot + " ha sido actualizado");
            }
}
```

9. Conexión de la lógica del negocio con el nivel de datos

A continuación tenéis que implementar la lógica del negocio apropiada para ser utilizada con la presentación realizada en el laboratorio "Separación entre Presentación y Lógica del Negocio".

```
package logicaNegocio;

public class LoginBD implements Login{

// Definir el código necesario para trabajar con una BD objectDB
 public boolean hacerLogin(String log, String pass, String tus){

// Comprueba si existe una cuenta con ese login, password y
// tipo de usuario en una BD ObjectDB
 }
public boolean add(String login, String password, String tus){

// Añade una nueva cuenta a la BD, devolviendo true si no existe
// o false si ya existía
}}
```

NOTA: Aunque es conveniente que dediquéis un tiempo a intentar resolver el problema, incluimos a continuación una solución.

SOLUCIÓN

```
package logicaNegocio;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import domain.Cuenta;

public class LoginODB implements Login {

	String fileName="passwords.odb";
	private  EntityManager  db;
	private EntityManagerFactory emf;

 public LoginODB() {

	 emf = Persistence.createEntityManagerFactory("objectdb:"+fileName);
	 db = emf.createEntityManager();
	 System.out.println("Base de datos abierta");
	 inicializarBD();
	}


 public void inicializarBD () {

	 add("Kepa","hola","Profesor");
	 add("Nerea","kaixo","Estudiante");
	}

 public boolean add(String login, String password, String tus){

	 if (db.find(Cuenta.class, login)!=null) return false;
	 db.getTransaction().begin();
	 Cuenta c = new Cuenta(login,password,tus);
	 db.persist(c);
	 db.getTransaction().commit();
	 return true;
	}

 public boolean hacerLogin(String login, String password, String tusuario){

	Cuenta c= db.find(Cuenta.class,login);
	if (c==null) return false;
	return (c.getPassword().equals(password)
		 && (c.getTusuario().equals(tusuario)));
	}
}
```

```java
package domain;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Cuenta {

	@Id
	String usuario;
	String password;
	String tusuario;

	public Cuenta(String usuario, String password, String tusuario) {
		super();
		this.usuario = usuario;
		this.password = password;
		this.tusuario = tusuario;
	}

	public String getUsuario() {
		return usuario;
	}

	public void setUsuario(String usuario) {
		this.usuario = usuario;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	public String getTusuario() {
		return tusuario;
	}

	public void setTusuario(String tusuario) {
		this.tusuario = tusuario;
	}
}
```