# Laboratorio de profundización de ObjectDB

## *Introducción*

En este laboratorio mostraremos alguna característica adicional de ObjectDB.

## *Objetivos*

- Saber implementar la bidireccionalidad en asociaciones
- Conocer el concepto de crear, actualizar y borrar objetos anidados
- Entender que JPQL es parecido a SQL pero con características OO
- Acceder de manera remota a la base de datos: uso de un servidor de BD

## *Pasos a seguir*

### 1. Descargar el proyecto con el que vamos a trabajar

Lo primero que hay que hacer es descargar y descomprimir el fichero **Lab2objectDB.zip** disponible en el sitio Moodle de la asignatura. A continuación se puede importar el proyecto en Eclipse y comprobar que tiene la siguiente estructura:

• **domain:** es un paquete con las clases y asociaciones de la BD **America.odb** que aparecen en la figura 1

• **gui** es un paquete con algunas clases de interfaz de usuario gráfico, que serán útiles para insertar datos en la base de datos (invocando a la lógica del negocio).

• **bussinesLogic** es un paquete con las clases de la lógica del negocio **FacadeInterface** y **FacadeImpl**, las cuales llaman a los métodos de la clases de acceso a datos.

• **dataAccess** es un paquete con las operaciones para almacenar objetos en la base de datos.

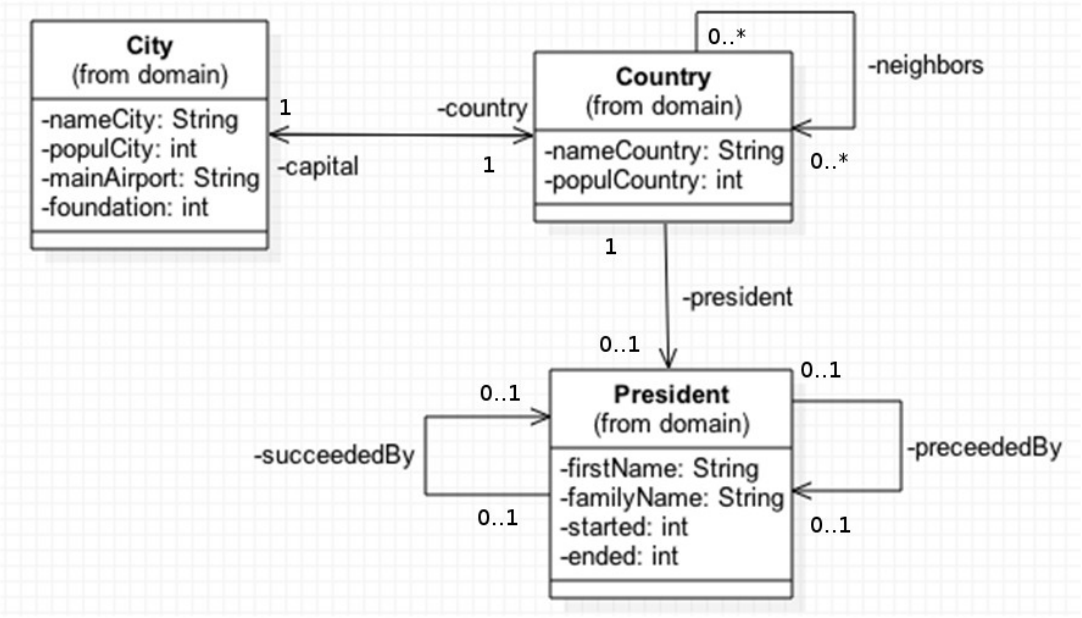
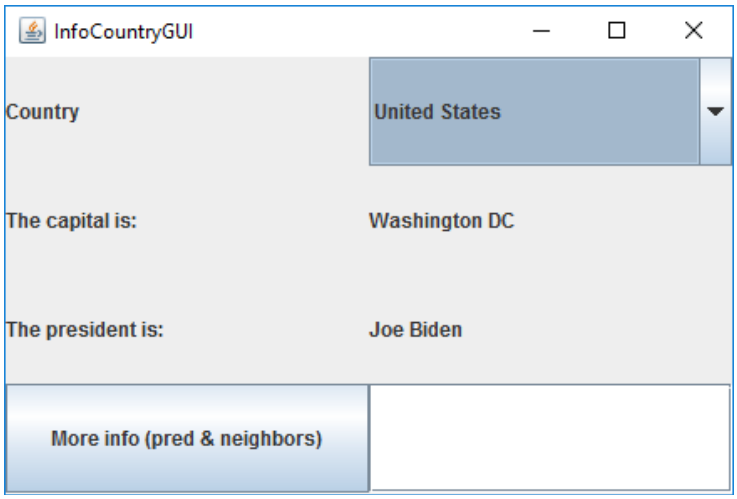


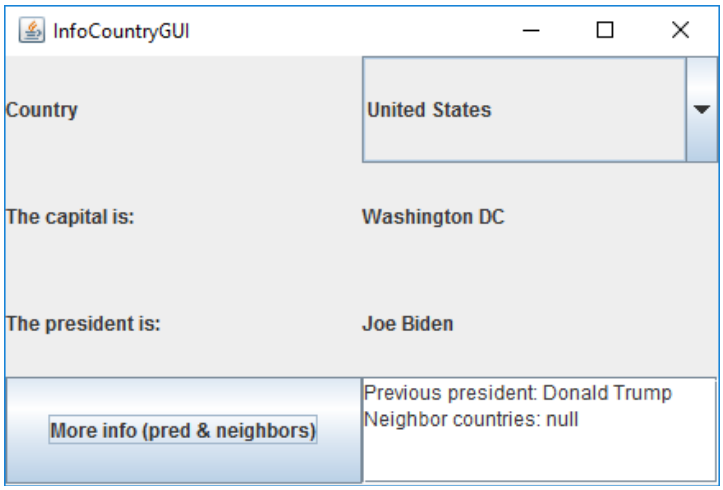

Figura 1: diagrama con las clases del dominio correspondientes a la base de datos America.odb

## 2. Consulta de objetos anidados.

La interfaz gráfica **`InfoCountryGUI`** se puede utilizar para visualizar información de diferentes países. Si seleccionamos “United States” veremos cuál es su capital y su presidente. Para el resto de países la información está incompleta.




Si pulsamos el botón “More info (pred & neighbors)” veremos que en el área de texto aparece información del predecesor del presidente actual e información de sus países vecinos.




Sin embargo, aunque en la base de datos hay información de sus países vecinos (se puede comprobar con el EXPLORER de objectDB), estos no aparecen en el área de texto, aunque sí la información de que Donald Trump es el predecesor del actual presidente. La razón de este comportamiento tiene que ver con cuándo se recuperan los objetos anidados dentro de otros objetos.

Java Persistence API (JPA), que es una de las APIs implementadas por objectDB, permite especificar mediante anotaciones las relaciones que existen entre clases, y establecer para las mismas diferentes propiedades.

En nuestro ejemplo, se pueden anotar las relaciones entre clases con las anotaciones `@OneToOne` y `@OneToMany`. Con la anotación `@OneToOne` se indica que una instancia de Country tiene una instancia de City (su `capital`) y una instancia de President (su `president`). Con la anotación `@OneToMany` se indica que una instancia de Country está asociada con varias instancias de Country (sus `neighbors`).

```
@Entity
public class Country {
	@Id
	private String nameCountry;
	@OneToOne
	private City capital;
	private int populCountry;
	@OneToMany
	private Set<Country> neighbors=new HashSet<Country>();
	@OneToOne
	private President president;
```

```
@Entity
public class City {
	@Id
	private String nameCity;
	@OneToOne
	private Country country;
	private int populCity;
	private String mainAirport;
	private int foundation;
```

```
@Entity @IdClass(President.class)
public class President {
	@Id
	private String firstName;
	@Id
	private String familyName;
	private int started;
	private AccessMode accessPower;
	private int ended;
	@OneToOne
	private President preceededBy;
	@OneToOne
	private President succeededBy;
```

**Nota:** con la anotación `@IdClass(President.class)` se indica que los atributos `firstName` y `familyName` forman la clave compuesta primaria de `President`.

No es estrictamente necesaria dicha anotación para definir una clave compuesta, pero sí es necesaria si se quisiera buscar un objeto de la clase por su clave primaria:

```
db.find(President.class, new President("Barak","Obama"))
```

En este punto hay que hacer lo siguiente:

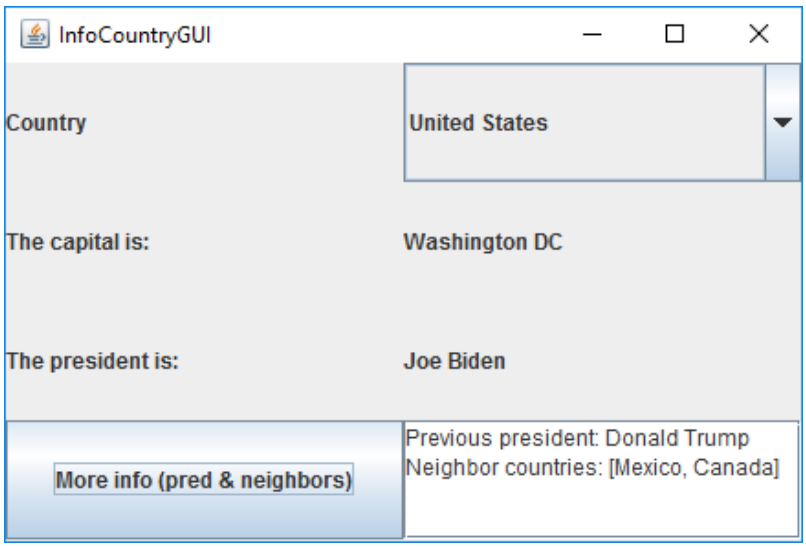
a) Poner las anotaciones anteriores y ejecutar de nuevo la clase **`InfoCountryGUI`**

b) Comprobar que el comportamiento es exactamente el mismo: al traer el objeto "United States", no se traen los objetos de sus países vecinos.

c) Modificar la clase Country para indicar que cuando se recupere una instancia de Country, se recuperen también las instancias de sus vecinos (aplicando la estrategia EAGER o "ansiosa").

```
@OneToMany(fetch=FetchType.EAGER)
private Set<Country> neighbors=new HashSet<Country>();
```

d) Comprobar que en la ejecución, ahora sí se informa de los países vecinos de EEUU:




e) Modificar la clase Country para indicar que, cuando se recupere la instancia de Country, no se recuperen las instancias de sus vecinos hasta que se necesiten (aplicando la estrategia LAZY o "perezosa")

```
@OneToMany(fetch=FetchType.LAZY)
private Set<Country> neighbors=new HashSet<Country>();
```

f) Comprobar que la ejecución funciona exactamente como al principio. Ello es debido a que la estrategia ***`LAZY`*** es la estrategia por defecto para el caso `@OneToMany`. Sin embargo, para el caso `@OneToOne`, la estrategia por defecto es la ***`EAGER`***, y por eso Donald Trump ha aparecido en todas las ejecuciones anteriores sin necesidad de haberlo establecido de manera explícita:

Nota: como curiosidad, aunque se pusiera la estrategia ***LAZY*** para `@OneToOne`, objectDB parece que no la implementa, por no ser obligatorio, según aparece en la documentación http://www.objectdb.com/api/java/jpa/OneToOne

```
@OneToOne(fetch=FetchType.LAZY)
private President president;
```

g) Descomentar el siguiente código en el método getAllCountries() de la clase DataAccess:

```
//     for (Country c: results) {
//           System.out.println(c+" with neighbors: "+c.getNeighbors());
//     }
```

Y usar la estrategia ***LAZY*** para recuperar los `neighbors` (en la clase Country)

```
@OneToMany(fetch=FetchType.LAZY)
private Set<Country> neighbors=new HashSet<Country>();
```

Volver a ejecutar la clase **`InfoCountryGUI`**, seleccionar “United States” y comprobar tras pulsar el botón “More info” que sí aparecen los países vecinos de EEUU.

Pregunta: ¿Por qué sucede?

Respuesta: con la estrategia ***LAZY*** se traen los países vecinos justo cuando se pide, esto es, cuando se ejecuta `c.getNeighbors()` dentro del método getAllCountries() correspondiente a la capa de acceso a datos perteneciente a la lógica del negocio, por lo que después de ejecutar `return results` dichos valores estarán cargados en los objetos de Country devueltos, y podrán mostrarse al pulsarse el botón de la clase **`InfoCountryGUI`** correspondiente a la capa de presentación.

Sin embargo, cuando no se ejecutaba `c.getNeighbors()` dentro del método getAllCountries(), aplicando la estrategia ***LAZY*** no se cargaban los países vecinos, y ya no se podían cargar después al haberse cerrado la conexión con la base de datos.

## 3. Creación de objetos anidados.

La interfaz gráfica **`NewCountriesGUI`** se puede utilizar para introducir nuevos países en la base de datos. El botón **Save country** tiene un "oyente" (listener) que se encargará de guardar los datos (country y capital), pero que no funciona ya que el método correspondiente al nivel de datos no está implementado. Se pide:

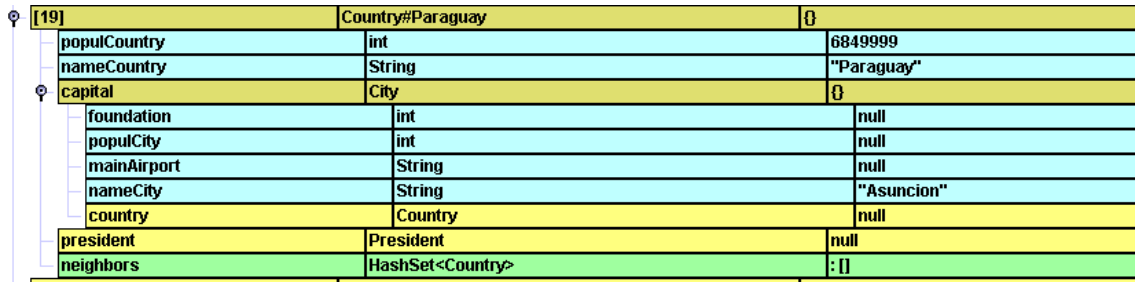
a) Implementar el método

**`DataAccess.saveCountryAndCity(countryName,countryPopul, capitalName,capitalPopul,airportName,foundationYear)`**

de la siguiente manera: abrir la conexión con la base de datos, crear una nueva transacción, crear una nueva instancia de Country (con los valores **`countryName`** y **`countryPopul`**) y almacenarla en `country`, crear una nueva instancia de City (con los valores **`capitalName, capitalPopul, airportName, foundationYear`** y el país `country`) y almacenarla en `city`, establecer `city` como la capital de `country`, dar persistencia al nuevo país **`db.persist`**(`country`), confirmar la transacción (commit), cerrar la conexión con la base de datos y devolver si ha habido error o no en la creación.

b) Probar la ejecución del método con los datos de algún nuevo país, como por ejemplo Paraguay. A excepción de las islas pequeñas que son naciones, faltan en la base de datos las siguientes naciones: Paraguay, Guyana y Uruguay. Hay que introducirlas en la BD utilizando la implementación realizada (ejecutando la clase **`NewCountriesGUI`**y la implementación de **`saveCountryAndCity`**), utilizando la información disponible en la Web (a no ser que os la sepáis de memoria).


c) Abrir la base de datos **America.odb** con el EXPLORER de objectDB y comprobar que hay un problema, ya que se le ha dado persistencia al nuevo país (Paraguay), que tiene asignada la capital (Asunción), pero que dicho objeto de City (Asunción) no se ha persistido de manera correcta. Sólo aparece el nombre de la ciudad, pero no el resto de datos.




d) Antes de solucionarlo, hay que borrar el país de la base de datos: hacer click derecho sobre el mismo en el EXPLORER => Delete => Cerrar el EXPLORER => Guardar cambios en la BD

Para realizar la persistencia de la capital correctamente se puede hacer de las siguientes dos maneras:

e) Dar la persistencia de manera explícita en el método **`saveCountryAndCity`** por medio de la instrucción **`db.persist`** (`city`). A continuación ejecutar **`NewCountriesGUI`** e introducir uno de los países que faltan (por ejemplo Paraguay) y comprobar en el EXPLORER que ahora sí se ha insertado correctamente la capital Asunción.

f) Realizar la persistencia de manera implícita o declarativa, utilizando la propiedad cascade en la anotación `@OneToOne` correspondiente a la asociación de la clase Country con City. Con el valor ***`PERSIST`*** se indica que cuando se dé persistencia al objeto de Country, se le dé persistencia al objeto capital de City también.

```
@OneToOne(cascade=CascadeType.PERSIST)
private City capital;
```

A continuación ejecutar **`NewCountriesGUI`** e introducir nuevos países que falten (por ejemplo Uruguay y Guayana) y comprobar en el EXPLORER que también se han insertado correctamente junto con sus capitales.

## 4. Actualización de objetos anidados.

Ahora usaremos una interfaz gráfica **ChooseNeighborsGUI** para insertar naciones vecinas, la cual corresponde a un caso de uso con el siguiente flujo de eventos:

1. **El sistema presenta los nombres de todas las naciones.**
2. **El usuario escoge la nación A.**
3. **El sistema borra la nación A de la lista y muestra la nueva lista.**
4. **El usuario escoge las naciones B1, B2, ..., Bn que desee establecer como vecinas de A**
5. **El sistema guarda la información de que B1, B2,…, Bn y A son naciones vecinas (Nota: que B1, B2,...,Bn son vecinas de A y su inversa, que A es vecina de B1,B2,...,Bn)**

Como en la interfaz anterior disponemos del botón etiquetado como **Add selected as neighbor countries** que se encarga de guardar los datos llamando para ello a un método de la capa de acceso a datos (DataAccess) cuya implementación es la siguiente:

```
public void assignNeighbors (Country home, List<Country> neighbors) {
	try {
		openDb();
		db.getTransaction().begin();
		Country homeDB = db.find(Country.class, home.getNameCountry());
		for (Country neighbor : neighbors) {
			Country neighDB = db.find(Country.class, neighbor);
			homeDB.addNeighbor(neighDB);
			neighDB.addNeighbor(homeDB);
		}
//		db.persist(homeDB);
//		db.persist(neighDB);
		db.getTransaction().commit();
	} catch (Exception e) {
		e.printStackTrace();
	} finally { closeDb(); }
}
```

En dicho método se hace lo siguiente: abrir la conexión con la BD, abrir una transacción, recuperar de la base de datos tanto el país seleccionado `home` como cada uno de los países `neighbor` que se desea asignar como vecino, asignar al país `home` cada uno de sus vecinos `neighbor`, y a cada uno de estos vecinos `neighbor` el país `home` (en realidad sus correspondientes objetos en la BD: `homeDB` y `neighDB`), y por último se confirman los cambios en la transacción (con el commit) y se cierra la BD.

La única duda en dicho código es si hay que indicar explícitamente el persist de `homeDB` y de `neighDB` (instrucciones que aparecen como comentarios en el método `assignNeighbors`). Se pide probarlo en el caso en que la clase Country establece la política ***PERSIST*** en `cascade` para el atributo `neighbors`, como se ve a continuación.

```
@OneToMany(fetch=FetchType.LAZY,cascade=CascadeType.PERSIST)
private Set<Country> neighbors=new HashSet<Country>();
```

Pregunta: ¿Cuál de las instrucciones persist es imprescindible? Para saberlo, se pueden probar las distintas posibilidades. Añadir, por ejemplo, los cinco países vecinos de Argentina, y verificar posteriormente en la BD que se han grabado y que Argentina ha sido a su vez añadida como país vecino de esos cinco países.

Solución: no hace falta ninguno de los persist porque como se mostró en el laboratorio 1 de objectDB, las actualizaciones se realizan de manera transparente. Aquellos objetos conectados con la BD dentro de la transacción que se hayan modificado se modificarán en la BD. Eso sí, para estar seguros de que esos objetos están conectados con la BD se han ejecutado previamente los `db.find(Country.class, ...)`

## 5. Borrado de objetos anidados.

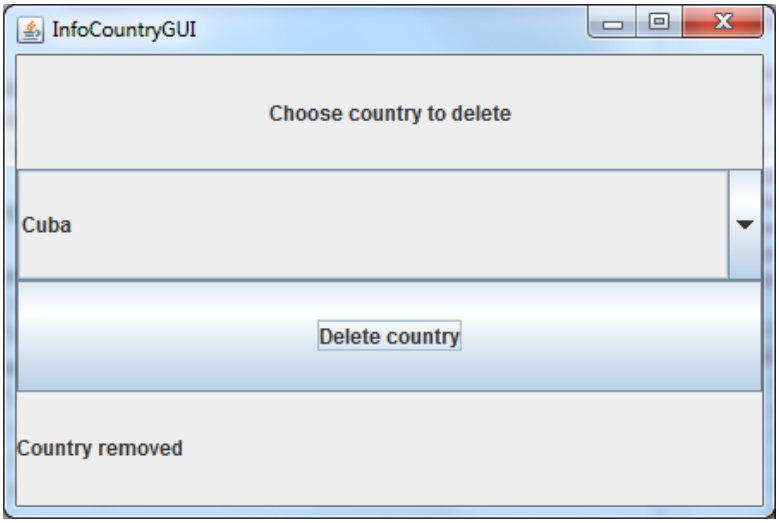
En este apartado se van a borrar países, por lo que es conveniente realizar una copia de la base de datos actual (la original con los nuevos países introducidos) antes de proceder a realizar algunos borrados. Realizar una copia de **America.odb** en el directorio donde se encuentra el proyecto, por ejemplo en el fichero **America-copia.odb**.

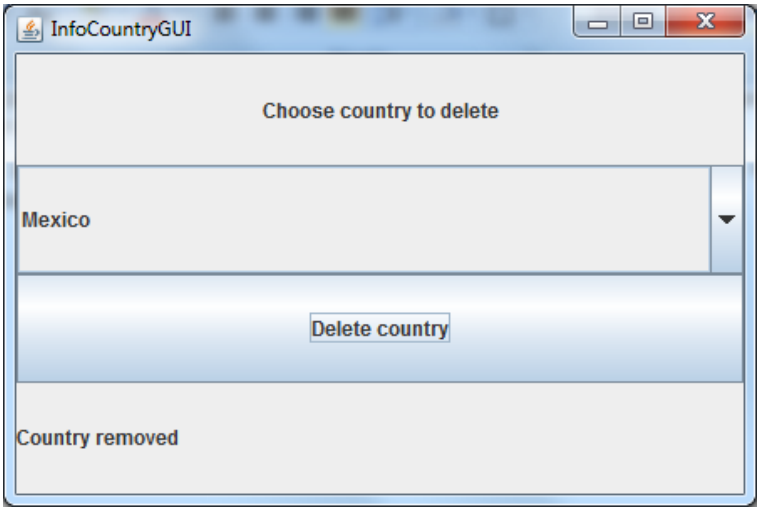
La interfaz gráfica **RemoveCountryGUI** permite eliminar un país. El botón **Delete country** tiene un "oyente" (listener) que invoca al método `removeCountry` que borra de la base de datos el país seleccionado. El método se encuentra en la clase DataAccess, y está implementado de la manera siguiente:

```
public boolean removeCountry (Country country){
	boolean res=true;
	try {
		openDb();
		db.getTransaction().begin();
		Country c=db.find(Country.class, country.getNameCountry());
		db.remove(c);
		db.getTransaction().commit();
		System.out.println("object removed "+country.getNameCountry());
	} catch (Exception e) {
		e.printStackTrace();
		res=false;
	} finally { closeDb(); }
	return res;
}
```

En dicho método se realiza lo siguiente: abrir la conexión con la BD, abrir una transacción, recuperar de la base de datos el país que se desea borrar `country`, borrar dicho país de la BD, confirmar el borrado de la transacción (commit) y cerrar la BD.

Borrar los países Cuba y Mexico. Para ello ejecutar la clase **`RemoveCountryGUI`** y pulsar el botón “Delete country” tras seleccionar uno a uno Cuba y Mexico en el JComboBox




Se puede comprobar con el EXPLORER de objectDB que las capitales no se han borrado correctamente de la BD. Por ejemplo, en el caso de Cuba, La Habana no se ha borrado. Aunque Cuba no aparece como objeto de Country, sí que está anidado dentro de La Habana (pero no con todos los valores, sino solamente el nombre del país).

| [8] | City#La Habana | {} |
|---|---|---|
| foundation | int | 1515 |
| populCity | int | 2100000 |
| mainAirport | String | "José Martí" |
| nameCity | String | "La Habana" |
| country | Country | {} |
| populCountry | int | null |
| nameCountry | String | "Cuba" |
| capital | City | null |
| president | President | null |
| neighbors | Set<Country> | null |

Encontrar la manera de que se borre en cascada la ciudad cuando se borre el país.

Solución:

```
@OneToOne(cascade=CascadeType.REMOVE)
private City capital;
```

Aunque, como ya estaba anotado con PERSIST, para mantener ambas se puede declarar como:

```
@OneToOne(cascade=CascadeType.ALL)
private City capital;
```

Si se vuelve a ejecutar **RemoveCountryGUI** (copiando previamente **America-copia.odb** a **America.odb**) para borrar Cuba y Mexico, se puede comprobar que las ciudades La Habana y Ciudad de Mexico desaparecen también.

Con respecto a los países vecinos tanto de Cuba como de Mexico, se puede comprobar que, aunque se ha borrado como país, Mexico aparece como país vecino de United States, aunque no como objeto con todos sus valores sino solamente su nombre.

| [19] | Country#United States | {} |
|---|---|---|
| populCountry | int | 320000000 |
| nameCountry | String | "United States" |
| capital | City#Washington DC | {} |
| president | President#[Ljava.lang.Object;@122d4cf | {} |
| neighbors | HashSet<Country> | 2 objects: [{}, {}] |
| [0] | Country#Canada | {} |
| [1] | Country | {} |
| populCountry | int | null |
| nameCountry | String | "Mexico" |
| capital | City | null |
| president | President | null |
| neighbors | Set<Country> | null |

Lo anterior no pasa con Cuba, no tanto por ser una isla, sino porque no tenía países vecinos en la BD

La solución al problema anterior no es tan sencilla. Alguien podría pensar que bastaría con establecer en la clase Country el borrado en cascada de los vecinos:

```
@OneToMany(fetch=FetchType.LAZY,cascade=CascadeType.REMOVE)
private Set<Country> neighbors=new HashSet<Country>();
```

Sin embargo, eso no es lo que se quiere: que al borrar Mexico se borre su país vecino (United States), sino que se borre Mexico de la lista de vecinos de United States, lo cual es bastante diferente.

Aprovechando que tenéis la copia **America-copia.odb** podéis comprobar el comportamiento anterior.

En realidad, lo que se desea realizar no es algo que pueda hacerse de manera automática (y tal vez no deba hacerse de manera automática). Así que tendríamos que borrar explícitamente el país de la lista de vecinos añadiendo el siguiente código en el método `public boolean removeCountry (Country country)` de DataAccess, antes de ejecutar `db.getTransaction().commit()`

```
for (Country nc: c.getNeighbors())
        nc.getNeighbors().remove(c);
```

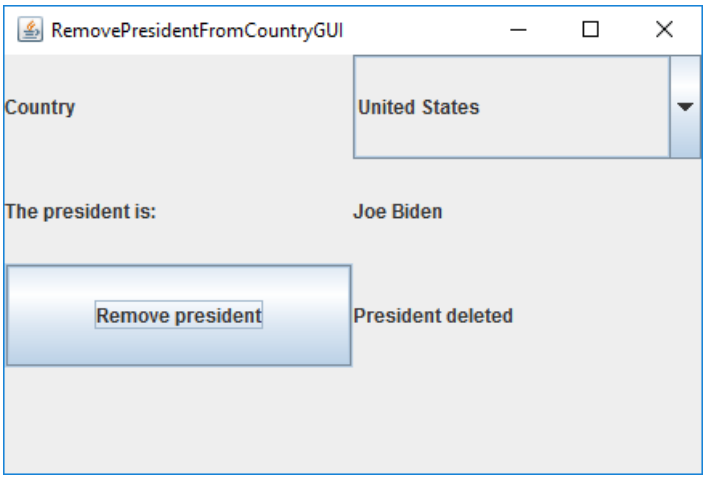
Es el desarrollador el que debe encargarse de esto, de la misma manera que debe controlar cuándo hay redundancia (por ejemplo, al añadir en dos sitios la información de que un país es vecino de otro).

## 6. Borrado de objetos referenciados/agregados en otros objetos.

En muchos casos, los objetos a borrar se suelen encontrar agregados en otros objetos que son los responsables de haberlos creado, y por tanto, deberían ser los responsables de borrarlos[1]. En nuestro caso, la clase Country va a ser la encargada de crear, modificar y borrar los datos de su presidente actual `president`

Para ello, se utiliza la clase **`RemovePresidentFromCountryGUI`**, que tras seleccionar el país (United States), permite borrar al presidente actual.

[1] Esto es así cuando se aplica un patrón de diseño GRASP conocido como CREADOR




El método que implementa ese borrado en DataAccess es:

```
public boolean removePresidentFromCountry (Country country){
	boolean res=true;
	try {
		openDb();
		db.getTransaction().begin();
		Country c=db.find(Country.class,country.getNameCountry());
		c.setPresident(null);
		db.getTransaction().commit();
	} catch (Exception e) {
		e.printStackTrace();
		res=false;
	} finally { closeDb(); }
	return res;
}
```

En este método se realiza lo siguiente: abrir la conexión con la BD, abrir una transacción, recuperar de la base de datos el país c cuyo presidente se desea quitar, eliminar la información del presidente de dicho país (poner a null), confirmar el borrado de la transacción (commit) y cerrar la BD. Nótese que la parte a analizar es la que invoca a `c.setPresident(null)`, la cual no borra la instancia del presidente sino que simplemente elimina (o anula) la información del presidente del país almacenado en c.

En este caso, usando el EXPLORER se puede comprobar en la BD que el presidente es `null`

| [21] | Country#United States | {} |
|---|---|---|
| populCountry | int | 320000000 |
| nameCountry | String | "United States" |
| capital | City#Washington DC | {} |
| president | President | null |
| neighbors | HashSet<Country> | 2 objects: [{}, {}] |

También se puede comprobar que la instancia de presidente sigue existiendo en la base de datos, aunque ya no esté agregada en ningún otro país. En realidad ese resultado es correcto porque no se ha hecho un “remove” de dicho objeto.

| [2] | President#domain.President@43d3f07d | {} |
|---|---|---|
| ended | int | 0 |
| started | int | 0 |
| familyName | String | "Biden" |
| firstName | String | "Joe" |
| accessPower | AccessMode | null |
| preceededBy | President#domain.President@5c86e02 | {} |
| succeededBy | President | null |

Sin embargo, al ser un presidente que no se encuentra referenciado en ningún otro país, podría interesar que fuera borrado de la BD. Para conseguir que dicha instancia que no se encuentra referenciada o agregada en ningún otro objeto se borre de la base de datos, hay que añadir `orphanRemoval=true` en la definición del atributo `president` de la clase Country.

```
@OneToOne(orphanRemoval=true)
private President president;
```

Ejecutar la clase **RemovePresidentFromCountryGUI** tras haber añadido la opción `orphanRemoval=true` en `president` y haber copiado de nuevo **America-copia.odb** a **America.odb** y comprobar que después se elimina también "Joe Biden" como objeto de President de la BD.

Notas:

1) Si "Joe Biden" fuera el presidente de otro país, no se borraría.

2) El atributo `orphanRemoval` se puede utilizar en relaciones `OneToMany`. Por ejemplo, si en vez de un presidente fueran varios:

```
@OneToMany(orphanRemoval=true)
private Set<President> presidents=new HashSet<President>();
```

Para eliminar al presidente `p` habría que eliminarlo del "HashSet" con la instrucción `presidents.remove(p)` (y no asignando `null`). Esto no eliminaría a `p` de la BD, sólo si `p` no estuviera en ninguna lista de presidentes entonces se eliminaría.

## 7. JPQL: el lenguaje OO para consultar la BD basado en SQL

En el primer laboratorio de objectDB mostramos algún ejemplo de consulta JPQL, y en este vamos a presentar algunas más. Son las siguientes:

1) Consulta que obtiene las ciudades con una población mayor que una cantidad `popul`. La implementación en JPQL de dicha consulta tiene la misma sintaxis que SQL. (con sus partes SELECT, FROM y WHERE)

```
"SELECT ci FROM City ci WHERE ci.populCity>=?1"
...setParameter(1,popul)
```

Se encuentra implementada en el siguiente método de la clase DataAccess:

```
public Collection<City> hugeCities(int popul)
```

2) Consulta que obtiene los países que tienen dichas ciudades grandes como capitales. La consulta JPQL necesita ahora hacer un JOIN entre COUNTRY y CAPITAL para obtener dichos países, y se asemeja bastante a SQL (excepto en `c.capital`, que no es una tabla):

```
"SELECT c FROM Country c join c.capital ci where ci.populCity>=?1"
...setParameter(1,popul)
```

Se encuentra implementada en el siguiente método de la clase DataAccess:

```
public Collection<Country> countriesWithHugeCapitals1(int popul)
```

3) Aunque JPQL permite utilizar JOIN, es muy posible que se puedan evitar como en este caso, teniendo en cuenta que a partir del objeto de City se puede obtener directamente su objeto `country`. En este caso sí es sintaxis SQL.

```
"SELECT ci.country FROM City ci where ci.populCity>=?1"
...setParameter(1,popul)
```

Se encuentra implementada en el siguiente método de la clase DataAccess:

```
public Collection<Country> countriesWithHugeCapitals2(int popul)
```

4) Otra manera de evitar el JOIN es aprovechar que desde el objeto Country se puede "navegar" hasta su objeto `capital` (de City):

```
"SELECT c FROM Country c where c.capital.populCity>=?1"
 ...setParameter(1,popul)
```

Se encuentra implementada en el siguiente método de la clase DataAccess:

```
public Collection<Country> countriesWithHugeCapitals3(int popul)
```

En esta última pregunta se utiliza `c.capital.populCity`, lo que claramente no se permite en SQL por ser una característica propia de un lenguaje Orientado a Objetos (OO) como JPQL

5) Por último, no hay que olvidar que habrá preguntas que no podrán expresarse directamente en JPQL (como por ejemplo: obtener las ciudades que tienen una población mayor que la de todos sus países vecinos), por lo que habrá que realizarlas directamente desde el lenguaje de programación.

La consulta anterior (obtener las ciudades con una población mayor que una cantidad `popul`) implementada de esta manera se encuentra en el método `countriesWithHugeCapitals4` de DataAccess, cuya implementación es esta:

```
public Collection<Country> countriesWithHugeCapitals4(int popul){

      ArrayList<Country> res=new ArrayList();
      try {
       openDb();
       TypedQuery<Country> query =
            db.createQuery("SELECT c FROM Country c", Country.class);
       List<Country> results = query.getResultList();

       for (Country pais : results) {
            if (pais.getCapital().getPopulCity()>=popul)
                  res.add(pais);
       }
      } catch(Exception e) {e.printStackTrace();}
      finally { closeDb(); }    return res;}
```

Todas estas consultas se pueden probar ejecutando la clase Main.

## 8. Acceso remoto a la base de datos: uso de un servidor de BD

Vamos a modificar la clase que implementa la capa de acceso a datos para que acceda a una base de datos remota. Para ello, habrá que hacer lo siguiente:

1) Lanzar el servidor de base de datos objectDB en la máquina que se desee (que actuará de máquina remota). De momento la máquina remota puede ser vuestra propia máquina.

   Tal y como se explica en la documentación de objectDB disponible en http://www.objectdb.com/java/jpa/tool/server, es suficiente con ejecutar el servidor (que es la clase java com.objectdb.Server que se encuentra en objectdb.jar) desde la línea de comandos para que acepte peticiones de conexión en un puerto (en este caso el 6136). Para pararlo, es similar, poniendo "stop" en vez de "start"

   ```
   > java -cp objectdb.jar com.objectdb.Server -port 6136 start
   ```

   Se puede lanzar el servidor de esta manera abriendo un intérprete de comandos (cmd) y ejecutándolo desde el directorio objectdb-X.X\bin, el mismo lugar donde se encuentra el EXPLORER de la BD (explorer.exe)

   Otra alternativa es ejecutar la clase `ObjectdbManagerServer` que os hemos proporcionado y que ejecuta exactamente ese comando.

2) Modificar el código de DataAccess para que cada vez que el cliente abra una conexión con la BD, lo haga con la base de datos remota. El resto de operaciones con la BD (consultas, inserciones, borrados, etc. no cambiarían)

   Para abrir una BD en modo local (situada en la misma máquina que el cliente) había que ejecutar lo siguiente:

```
      emf = Persistence.createEntityManagerFactory(dbName);
      db = emf.createEntityManager();
```

Para hacerlo en modo remote, tan sólo hay que modificar la información que se le envía al *createEntityManagerFactory* para que no sea sólo el nombre de la BD (almacenado en `dbName`) sino el de la máquina, puerto e información de usuario y password (obligatoria cuando se trabaja en modo Cliente/Servidor en objectDB).

Este es el nuevo código de openDB en DataAccess:

```
private void openDb() {
 if (!local) {
       Map<String, String> properties = new HashMap<String, String>();
       properties.put("javax.persistence.jdbc.user", user);
       properties.put("javax.persistence.jdbc.password", pass);
       emf = Persistence.createEntityManagerFactory(
                    "objectdb://"+ip+":"+port+"/"+dbName, properties);
       db = emf.createEntityManager();
 }
 else {
       emf = Persistence.createEntityManagerFactory(dbName);
       db = emf.createEntityManager();
 }
System.out.println("Database opened");
}
```

Que utiliza los siguientes valores (definidos como atributos de la clase DataAccess):

```
private String dbName = "America.odb";
private boolean local=false;
String ip="localhost";
int port=6136;
String user="admin";
String pass="admin";
```

En este punto se puede:

Ejecutar ahora de nuevo la funcionalidad anterior: **`InfoCountryGUI`**, **`NewCountriesGUI`**, **`ChooseNeighborsGUI`**, Main,… Si no funciona correctamente es porque el servidor de BD no está trabajando con la BD **America.odb** correcta.

Para ello hay que dejar la BD en el lugar donde objectDB la busca, esto es, en un directorio "db" situado, o bien en el directorio donde se encuentra el proyecto (si se ha lanzado con `ObjectdbManagerServer`) o bien en el directorio donde se encuentra objectdb-2.6.9_09 (si se ha lanzado el servidor desde la línea de comandos).

Nota: un truco para saber dónde está podría ser ejecutar **`NewCountriesGUI`** y buscar la nueva BD **America.odb** para ver dónde la ha creado objectDB.

Probar a ejecutar la aplicación con el servidor realmente remoto (y no localhost), poniendo el número de ip de otra máquina de algún compañero/a y modificándolo en DataAccess

```
String ip="158.227...";
```