CICLO FORMATIVO DE GRADO SUPERIOR

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

U.T. 3

Acceso a bases de datos orientadas a objetos

Acceso a datos

Curso: 2º

Rogelio García de la Reina Tejado[1]

rogeliogarcia@iesgregorioprieto.com

1 Perfil en Linkedin

Programación

v.1.1

Octubre - 2021

Rogelio García de la Reina Tejado

ÍNDICE

1 - Introducción

Los modelos de bases de datos tradicionales (relacional, red y jerárquico) han sido capaces de satisfacer con éxito las necesidades, en cuanto a bases de datos, de las aplicaciones de gestión tradicionales. Sin embargo, presentan algunas deficiencias cuando se trata de aplicaciones más complejas o sofisticadas. Las bases de datos orientadas a objetos se crearon para tratar de satisfacer las necesidades de estas nuevas aplicaciones.

La orientación a objetos ofrece flexibilidad para manejar algunos de estos requisitos y no está limitada por los tipos de datos y los lenguajes de consulta de los sistemas de bases de datos tradicionales. Una característica clave de las bases de datos orientadas a objetos es la potencia que proporcionan al diseñador al permitirle especificar tanto la estructura de objetos complejos, como las operaciones que se pueden aplicar sobre dichos objetos. Otro motivo para la creación de las bases de datos orientadas a objetos es el creciente uso de los lenguajes orientados a objetos para desarrollar aplicaciones.

Las bases de datos orientadas a objetos se han diseñado para que se puedan integrar directamente con aplicaciones desarrolladas con lenguajes orientados a objetos, habiendo adoptado muchos de los conceptos de estos lenguajes.

El modelo estándar ODMG

Este modelo especifica los elementos que se definirán, y en qué manera se hará, para la consecución de persistencia en las bases de datos orientadas a objetos que soporten el estándar. Consta de un lenguaje de definición de objetos, ODL, que especifica los elementos de este modelo.

Un grupo de representantes de la industria de las bases de datos formaron el ODMG (Object Database Management Group) con el propósito de definir estándares para los SGBD orientados a objetos. Este grupo propuso un modelo estándar para la semántica de los objetos de una base de datos. Su ultima versión, ODMG 3.0, apareció en enero de 2000. Una vez finalizado su trabajo, el grupo se disolvió en 2001. En 2006, un nuevo grupo de trabajo, el Object Management Group (OMG), empezó a trabajar en la cuarta generación del estándar de las bases de datos orientadas a objetos, que reflejase los cambios tecnológicos producidos, a partir de 2001, en las bases de datos orientadas a objetos. Los trabajos de este grupo finalizaron en 2008, sin los resultados esperados. A partir de esa fecha no se ha vuelto a trabajar sobre este tema.

Componentes

- Modelo de objetos: El modelo de objetos ODMG permite que tanto los diseños, como las implementaciones, sean portables entre los sistemas que lo soportan. Dispone de las siguientes primitivas de modelado:
    - Los componentes básicos de una base de datos orientada a objetos son los objetos y los literales. Los objetos y los literales se categorizan en tipos.
    - Cada tipo tiene un dominio específico compartido por todos los objetos y literales de ese tipo. Los tipos también pueden tener comportamientos. En el sentido práctico, un tipo puede ser una clase de la que se crea un objeto, una interface o un tipo de datos para un literal (por ejemplo, integer). Los objetos tienen propiedades, operaciones y las relaciones que tienen con otros objetos.

    - Una base de datos es un conjunto de objetos almacenados que se gestionan de modo que puedan ser accedidos por múltiples usuarios y aplicaciones. La definición de una base de datos está contenida en un esquema que se ha creado mediante el lenguaje de definición o especificación de objetos ODL (Object Definition Language) .

- Lenguaje de Especificación de objetos (ODL): Es un lenguaje de especificación para definir tipos de objetos para sistemas compatibles con ODMG. ODL es el equivalente del DDL (lenguaje de definición de datos) de los SGBD tradicionales. Define los atributos y las relaciones entre tipos.
- Lenguaje de Consulta de objetos (OQL): Es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras. Está basado en SQL-92, proporcionando un súperconjunto de la sintaxis de la sentencia SELECT. OQL no posee primitivas para modificar el estado de los objetos ya que las modificaciones se pueden realizar mediante los métodos que éstos poseen. La sintaxis básica de OQL es una estructura SELECT...FROM...WHERE..., como en SQL.

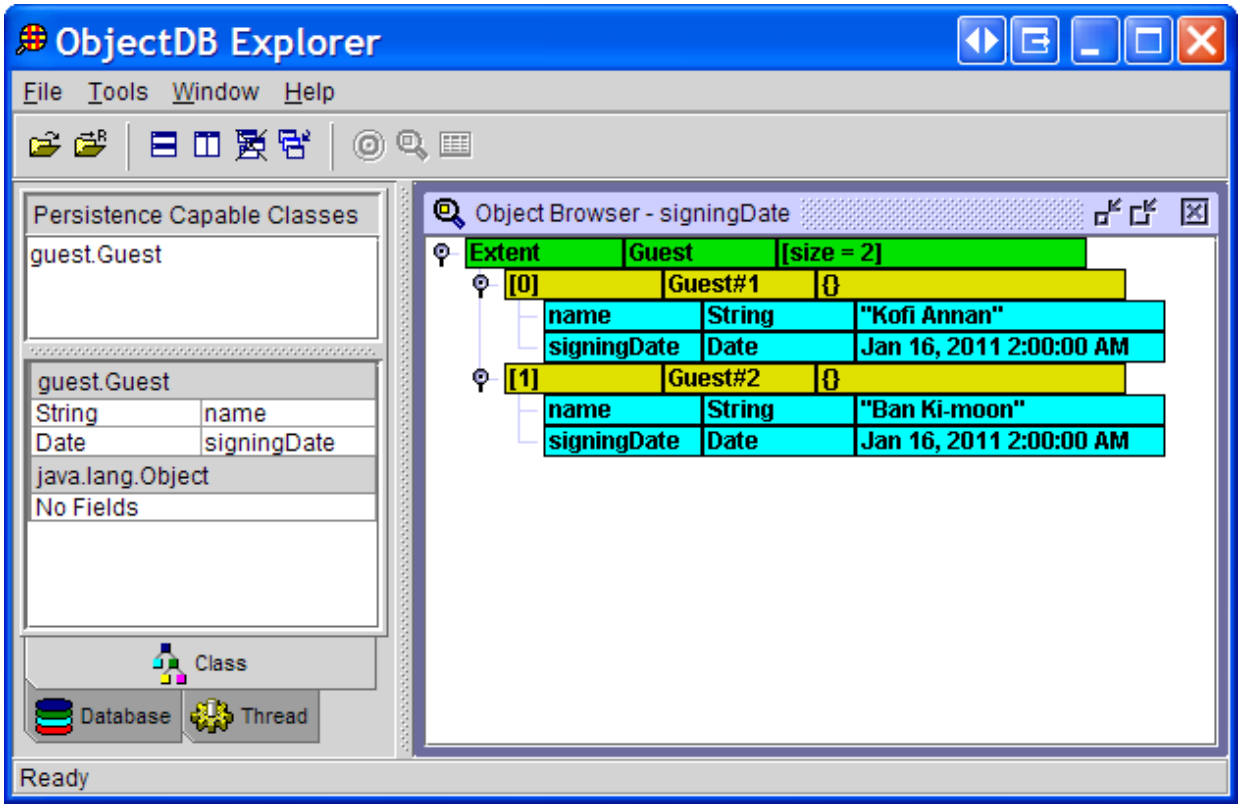
2 - ObjectDB

ObjectDB es una base de datos orientada a objetos para Java. Se puede utilizar en modo cliente-servidor y en modo embebido. A diferencia de otras bases de datos orientadas a objetos, *ObjectDB* no proporciona su propia API propietaria. Por lo tanto, el trabajo con *ObjectDB* requiere el uso de una de las dos API estándar de Java - JPA o JDO. Ambas APIs están incorporadas en *ObjectDB*, por lo que no es necesario un software ORM (*Object/Relational Mapping*) intermedio.

Utilidades y herramientas

Database Explorer

Es una aplicación Java incluida en *ObjectDB* que permite gestionar bases de datos ObjectDB. Permite editar el contenido de las bases de datos y ejecutar consultas JPQL y JDOQL.



El *ObjectDB Explorer* está incluido en el fichero *explorer.jar* incluido en el directorio bin del *ObjectDB*. Para ejecutarlo se debe utilizar el siguiente comando:

> *> java -jar explorer.jar*

Alternativamente, en Windows se puede ejecutar el *explorer.exe* (Windows) o explorer.sh (Linux/Unix). Ambos incluidos en el mismo directorio.

ObjectDB Server

Es una herramienta que permite gestionar bases de datos *ObjectDB* en un proceso separado, haciéndolas accesibles desde distintas aplicaciones locales o remotas. Su rendimiento es menor que en modo embebido. En aquellas bases de datos diseñadas para ser accedidas desde una única aplicación, es preferible utilizarlas en modo embebido, corriendo así en el mismo proceso de dicha aplicación.

Para ejecutar el servidor de *ObjectDB*, primero hay que indicarle a Java dónde buscarlo con la siguiente orden:

> java -cp [directorio]\objectdb.jar com.objectdb.Server

A partir de ese momento se puede ejecutar el servidor con diferentes opciones:

- *Iniciar: > java com.objectdb.Server -port 8888 start*
- *Parar: > java com.objectdb.Server stop*
- *Reiniciar: > java com.objectdb.Server restart*

En Windows también se puede gestionar el servidor con el comando *server.exe*.

ObjectDB Doctor

ObjectDB también incluye dos servicios:

- Diagnosis y validación de ficheros de bases de datos:
  > java -cp objectdb.jar com.objectdb.Doctor my.odb
- Reparación de bases de datos:

> java -cp objectdb.jar com.objectdb.Doctor old.odb new.odb

3 - JPA (Java Persistence API)

Como ya se indicó en la sección anterior *ObjectDB* implementa JPA de forma nativa, por tanto, no requiere de ningún ORM intermedio.

Definición de clases

Para definir una clase (utilizando JPA) se necesita definir una *entity class* lo cual permitirá representar objetos en la base de datos:

```
@Entity
public class NombreClase{
}
```

Si una clase incluye atributos de algún tipo definido por el propio usuario, estos se almacenarán en la base de datos con entidad propia si en la definición de su clase incluye la anotación @*Entity*. Si queremos que estos objetos se almacenen como parte de la clase que los contiene, deberemos añadirles la anotación @*Embeddable*:

```
@Embeddable
public class Address {
   String street;
   String city;
   String state;
   String country;
   String zip;
}
```

Al utilizar clases embebidas se optimiza el espacio ocupado en la base de datos y se mejora el rendimiento. La desventaja principal es que no podemos acceder directamente a ellas (no tienen clave primaria) y no podemos realizar consultas directas, solo a través de las entidades que las contienen.

El almacenamiento de un objeto en la base de datos NO guarda los métodos ni el código, sino solo el estado del objeto y en particular solo los datos de tipo persistente. Por defecto, cualquier campo que no es declarado como estático (*static*) o transitorio (*transient*) es persistente (*persistent*). *ObjectDB* soporta identificadores implícitos, por lo que no es necesario definir claves primarias (*Primary Keys*).

Si deseamos especificar que uno de los atributos de la clase será la clave primaria, deberemos añadir la anotación *@Id* al atributo (o atributos) que queramos actúe como clave primaria. Si queremos, que además, el valor de dicha clave se genere de forma automática deberemos también incluir la anotación @*GeneratedValue*:

```
@Id @GeneratedValue long id;
```

Conexión a la base de datos

La conexión mediante JPA a la base de datos requiere una interfaz *EntityManager*. Aquellas operaciones que modifican la base de datos requieren una instancia de tipo *EntityTransaction*. JPA requiere la definición de una unidad de persistencia en un archivo XML con el fin de generar una *EntityManagerFactory* para establecer la conexión, pero con *ObjectDB* basta con proporcionarla ruta donde se encuentran almacenados los objetos:

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("basedatos.odb");
EntityManager em = emf.createEntityManager();
```

Para cerrar el gestor de objetos (*EntityManager*) y la base de datos:

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

Tipos de datos persistentes

El término tipos persistentes se refiere a todos aquellos tipos de datos que pueden ser almacenados en una base de datos. *ObjectDB* puede almacenar todos los tipos persistentes de *JPA*:

- Clases definidas por el usuario: clases de tipo entidad (*Entity*), superclases asociadas y clases de tipo embebido.
- Tipos de datos simples de Java: tipos primitivos, *wrappers*, cadenas (*String*), fechas (*Date*), etc.
- Tipos de Colecciones (*Collections*), diccionarios (*Maps*) y arreglos(*Arrays*).
- Otros tipos: enumeradores (*Enum*) y tipos serializables definidos por el usuario o por el sistema.

Operaciones CRUD

El concepto de persistencia implica el hecho de realizar operaciones CRUD sobre nuestros objetos. CRUD (Create, Read, Update and Delete) son operaciones básicas de creación, lectura, actualización y eliminación de un registro u objeto en la base de datos.

Crear o almacenar objetos en la base de datos (Create)

Las operaciones que modifican el contenido de la bases de datos requieren una transacción activa. Mediante el método *persist*, asociado a una transacción activa generada por una instancia de *EntityManager*, es posible realizar el almacenamiento de objetos persistentes:

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("basedatos.odb");
EntityManager em = emf.createEntityManager();
Empleado empleado = new Empleado("Sergio", "Pérez");
em.getTransaction().begin();
em.persist(empleado);
em.getTransaction().commit();
em.close();
emf.close();
```

El nuevo objeto de la entidad Empleado es almacenado cuando la transacción es autorizada (*commit*). Si un objeto tiene a otro objeto como atributo (embebido), ese objeto también será almacenado en la base de datos al almacenar el primero, pero no podrá ser compartido por otros objetos. Cada objeto deberá tener y almacenar sus propios objetos embebidos. Si vamos a almacenar muchos objetos de una vez, podemos utilizar los métodos *clear* y *flush* para optimizar el uso de memoria en transacciones muy grandes.

Leer objetos de la base de datos (Read)

Podemos leer objetos de la base de datos utilizando la clave primaria:

```
Employee employee = em.find(Employee.class, 1);
```

Si el objeto ha sido leído previamente, podemos volver a cargarlo de la BD con:

```
em.refresh(empleado);
```

Actualización y borrado de objetos (Update y Delete)

Una vez cargado en memoria un objeto desde la base de datos, la actualización de dicho objeto en la base de datos se producirá de forma automática si la modificación del objeto se realiza dentro de una transacción activa. El borrado se realizará de forma análoga:

```
Employee employee = em.find(Employee.class, 1);
em.getTransaction().begin();
employee.setNickname("Joe the Plumber");
em.remove(employee);
em.getTransaction().commit();
```

El objeto es actualizado físicamente en la base de datos al hacer el *commit*.

JPA Query

El *JPQL* puede ser considerado la versión de *SQL* para las bases de datos orientadas a objetos. La sintaxis de *JPQL* es muy similar a la del *SQL*. La principal diferencia es que *SQL* trabaja con tablas y *JPQL* con clases y objetos.

CONSULTAS

EJECUTAR UNA CONSULTA QUE DEVUELVE MÁS DE UN RESULTADO:

```
String consulta = "SELECT c FROM Country c";
TypedQuery<Country> query = em.createQuery(consulta, Country.class);
List<Country> results = query.getResultList();
```

Esta lista puede ser recorrida con un bucle *foreach* o con un iterador.

EJECUTAR UNA CONSULTA QUE DEVUELVE UN ÚNICO RESULTADO:

```
String consulta = "SELECT COUNT(c) FROM Country c";
TypedQuery<Long> query = em.createQuery(consulta, Long.class);
long countryCount = query.getSingleResult();
```

EJECUTAR UNA CONSULTA QUE DEVUELVE SOLO ALGUNOS ATRIBUTOS DE UN OBJETO:

```
TypedQuery<Object[]> query = em.createQuery(
"SELECT c.name, c.capital.name FROM Country AS c", Object[].class);
List<Object[]> results = query.getResultList();
for (Object[] result : results) {
        System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

BORRADO Y ACTUALIZACIÓN

```
int count = em.createQuery("DELETE FROM Country").executeUpdate();
int count = em.createQuery("UPDATE Country SET area = 0").executeUpdate();
```

4 - JDO (Java Data Objects)

ObjectDB también implementa el estándar *JDO*, que permite hacer persistentes los objetos. Además, las aplicaciones que utilizan *ObjectDB* pueden cambiar a otra base de datos con soporte *JDO* sin tener que tocar el código.

Persistencia de clases

Para poder hacer persistentes los objetos debemos declarar la clase, a la que pertenecen dichos objetos, como persistente. Para ello, deberemos:

- Declararla en el fichero de metadatos.
- Incluir un constructor sin parámetros.
- Implementar el interfaz *javax.jdo.spi.PersistenceCapable*.

Datos persistentes

Por defecto, todos los atributos, de una clase persistente, no definidos como *static*, *final* o *transient* serán persistentes.

También son persistentes los tipos:

- Simples: *boolean*, *byte*, *short*, *char*, *int*, *long*, *float* y *double*.
- La clase *String* y todos los envoltorios (*Byte*, *Short*, *Integer*,...).
- Las clases: *Date*, *Locale*, *HashSet*, *TreeSet*, *ArrayList*, *LinkedList*, *Vector*, *HashMap*, *TreeMap* y las interfaces *Collection*, *Set*, *List* y *Map*.
- *Java.math.BigInteger* y *java.math.BigDecimal.*
- Cualquier *array* o matriz de alguno de los tipos anteriores.

Fichero de metadatos (package.jdo)

El fichero de metadatos contiene un documento XML en el que se declaran las clases persistentes.

```
<jdo>
  <package name="">
    <class name="A" />
  </package>
  <package name="test">
    <class name="B" />
    <class name="C" />
    <class name="D" />
  </package>
</jdo>
```

Metadatos de los atributos

Normalmente en el fichero de metadatos solo se especifican las clases. Solamente se incluyen los atributos cuando es necesario cambiar el comportamiento por defecto:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="test">
    <class name="A">
```

```
    <field name="f0" persistence-modifier="persistent" />
    <field name="f1" persistence-modifier="none" />
    <field name="f2" persistence-modifier="transactional" />
    <field name="f3" default-fetch-group="true" />
    <field name="f4" default-fetch-group="false" />
    <field name="f5" embedded="true" />
    <field name="f6" embedded="false" />
    <field name="f7" null-value="exception" />
    <field name="f8" null-value="none" />
    <field name="f9" null-value="default" />
   </class>
  </package>
</jdo>
```

Los atributos que pueden acompañar al nodo *field* (para cambiar su comportamiento) son:

- **persistence-modifier(persistent|none|transactional)**: Para indicar si el atributo debe ser persistente o no.
- **default-fetch-group (true | false)**: Para indicar si el atributo debe ser tratado en un grupo con otros campos.
- **embedded (true | false)**: Para indicar si un atributo debe ser almacenado embebido en el objeto que lo contiene o almacenado de forma separada con su propio ID. Si no se incluye, el valor por defecto es true.
- **null-value (exception | none | default)**: Indica cuando un atributo puede aceptar un valor nulo o no. Si se indica *exception*, si el valor del atributo es nulo se lanzará una excepción. Si se establece el valor none, se admitirán valores nulos y con default se almacenará el valor por defecto para dicho tipo (0 para enteros , comillas para String, etc.).

Vectores, colecciones y mapas

Los atributos de tipo colección, pueden incluir elementos en el XML, según su tipo, éste será diferente:

```
<jdo>
 <package name="test">
  <class name="B">
   <field name="f0" embedded="true">
    <array embedded-element="true" />
   </field>
   <field name="f1" embedded="true">
    <collection embedded-element="true" />
   </field>
   <field name="f2" embedded="true">
    <map embedded-key="true" embedded-value="true" />
   </field>
  </class>
 </package>
</jdo>
```

Estos nodos pueden incluir los siguientes atributos:

- **embedded-element**: Indica si el atributo de colección debe ser almacenado como un objeto embebido del objeto al que pertenece.
- **Embedded-key *y* embedded-value (true | false)**: Para indicar si las claves y valores de un mapa (map) deben ser almacenados como objetos embebidos o no.

Definición de índices

Para mejorar los tiempos de respuesta en las consultas podemos definir índices:

```
<jdo>
  <package name="test">
    <class name="A">
      <extension vendor-name="objectdb" key="index" value="field0" />
      <extension vendor-name="objectdb" key="unique-index" value="field1" />
    </class>
  </package>
</jdo>
```

Para definir índices de atributos embebidos (no tienen identificador de objeto), deberemos hacerlo de la siguiente forma:

```
<extension vendor-name="objectdb" key="index" value="field0.x" />
```

Conexión a la base de datos

La mayor parte de las aplicaciones requieren de múltiples conexiones a la base de datos durante su tiempo de vida. En el caso de las aplicaciones web es muy común establecer una conexión con la base de datos por cada petición web. Esto no es muy recomendable, ya que se podría superar el número de máximo de conexiones permitidas a la base de datos, además de sobrecargar el servidor de la base de datos de forma innecesaria.

Es habitual, en este tipo de aplicaciones trabajar con un *pool* de conexiones, de forma que las conexiones abiertas por la base de datos puedan ser reutilizadas una vez servida una petición, sin tener que cerrar dicha conexión, que será utilizada para servir la siguiente petición.

Obtener una instancia de la PersistenceManagerFactory

En *JDO* la clase *javax.jdo.PersistenceManagerFactory* se encarga de crear las conexiones cuando se requiere y de gestionar de forma automática el *pool* de conexiones. De esta forma, nosotros solo tenemos que pedirle una nueva conexión cuando los necesitamos y será el *PersitenceManagerFactory* el que se encargue de gestionarlas.

```
import java.util.Properties;
import javax.jdo.*;
try {
   Properties properties = new Properties();
     properties.setProperty(
   "javax.jdo.PersistenceManagerFactoryClass", "com.objectdb.jdo.PMF");
  properties.setProperty(
    "javax.jdo.option.ConnectionURL", "objectdb://localhost/my.odb");
  properties.setProperty("javax.jdo.option.ConnectionUserName", "john");
  properties.setProperty("javax.jdo.option.ConnectionPassword", "itisme");
   PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);
   PersistenceManager pm = pmf.getPersistenceManager();
```

```
}catch (JDOException e) {//Procesar error
}finally {
        if (!pm.isClosed()) pm.close();
}
```

También es posible obtener una instancia de la conexión (PersistenceManager) sin utilizar el *pool* de conexiones:

```
PersistenceManager pm = Utilities.getPersistenceManager("local.odb");
```

Fichero de propiedades

Las propiedades de la conexión también pueden ser obtenidas del fichero **jdo.properties**, que debe estar en el mismo directorio que la clase:

```
javax.jdo.PersistenceManagerFactoryClass=com.objectdb.jdo.PMF
javax.jdo.option.ConnectionURL=local.odb
```

El siguiente código muestra como establecer la conexión, cargando los datos de ésta del fichero anterior:

```
import java.io;
import java.util.Properties;
import javax.jdo.*;

 InputStream in = getClass().getResourceAsStream("jdo.properties");

try {
   Properties properties = new Properties();
   properties.load(in);
   PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(properties);
 }
 finally {
   in.close();
 }
 //Obtenemos la conexión
PersistenceManager pm = pmf.getPersistenceManager();
//Cerramos la conexión
if (!pm.isClosed()) pm.close();
```

Trabajar con transacciones

Para trabajar con transacciones con JDO haremos uso de la clase javax.jdo.Transaction:

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tr = pm.currentTransaction();
try {
        tr.begin();
        // Operations that modify the database should come here.
        tr.commit();
} finally {
        if (tr.isActive()) tr.rollback();
        if (!pm.isClosed()) pm.close();
}
```

Operaciones CRUD

Cuando creamos un nuevo objeto en una aplicación JDO, no éste no se hará persistente hasta que lo indiquemos en una transacción:

```
Person person = new Person("George", "Bush");
Address address = new Address("White House");
person.setAddress(address);
pm.currentTransaction().begin();
pm.makePersistent(person);
pm.currentTransaction().commit();
```

IDENTIFICADORES

ObjectDB asigna un identificador único a cada objeto que se hace persistente. El identificador del primer objeto es siempre 1, el del segundo 2 y así sucesivamente. Podemos obtener el identificador de un objeto de dos formas:

```
Object oid = pm.getObjectId(obj);
Object oid = JDOHelper.getObjectId(obj);
//Para mostrar el identificador del objeto
System.out.println(oid.toString());
```

OBTENER UN OBJETO DE LA BASE DE DATOS

Podemos obtener un objeto de la base de datos simplemente conociendo su identificador:

```
Object oid = pm.newObjectIdInstance(Person.class, “1”);
Person person = (Person)pm.getObjectById(oid);
```

OBTENER TODOS LOS OBJETOS DE UNA CLASE

También podemos obtener todos los objetos de una clase, almacenados en la base de datos, y recorrerlos con un iterador:

```
Extent extent = pm.getExtent(Person.class, false);
java.util.Iterator itr = extent.iterator();
try {
        while (itr.hasNext()) {
                Person person = (Person)itr.next();
                 System.out.println(person.getName());
        }
} finally {
         extent.close(itr); // closes a specified iterator
}
```

El segundo parámetro del método *getExtent*, permite indicar si se deben incluir las instancias de las subclases.

MODIFICAR Y BORRAR OBJETOS PERSISTENTES

Para modificar un objeto persistente, simplemente, se modifica el atributo dentro de una transacción:

```
Object oid = pm.newObjectIdInstance(Person.class, “1”);
Person person = (Person)pm.getObjectById(oid);
pm.currentTransaction().begin();
```

```
person.address = new Address("Texas"); //actualizamos
pm.deletePersistent(person);            //borramos
pm.currentTransaction().commit();       //hacemos persistentes los cambios
```

Cuando borramos un objeto de la base de datos, los objetos embebidos son borrados automáticamente. Los no embebidos referenciados por los objetos borrados no son eliminados. Si se quieren eliminar tendrán que ser eliminado explícitamente. Una forma elegante de hacerlo es utilizar una *InstanteCallbacks*.

```
class Line implements InstanceCallbacks {
  private Point p1, p2;
  public Line(Point p1, Point p2) {
    this.p1 = p1;
    this.p2 = p2;
  }
  public void jdoPreDelete() {
        PersistenceManager pm = JDOHelper.getPersistenceManager(this);
        pm.deletePersistent(p1);
        pm.deletePersistent(p2);
  }
}
```

JDOQL (JDO Query Language)

En la sección anterior se detalló la forma de obtener objetos de la base de datos utilizando su identificador y como obtener todos los objetos de una clase determinada. Si necesitamos ser más selectivos tendremos que hacer uso del JDOQL. Este lenguaje de consulta es similar al SQL.

CONSULTAS

En el siguiente ejemplo se muestra una consulta con un filtro, dónde el resultado es ordenado por un atributo de la clase:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
query.setOrdering("this.age ascending");
Collection result = (Collection)query.execute();
try {
   Iterator itr = result.iterator();
   while (itr.hasNext())
     System.out.println(itr.next());
} finally {
   query.close(result);
}
```

BORRADO

```
Query query = pm.newQuery("SELECT FROM Person WHERE age>18");
Long number = (Long)query.deletePersistentAll();
```

ACTUALIZACIÓN

JDO no permite la actualización masiva mediante consulta (al menos en objectdb).