Capítulo 15. Persistencia

A. J. Pérez

Persistencia

Se entiende por persistencia (en programación) como la acción de preservar la información de un objeto de forma permanente (guardar), pero a su vez también se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada.

En el caso de persistencia de objetos, la información que persiste en la mayoría de los casos son los valores que contienen los atributos en ese momento, no necesariamente la funcionalidad que proveen sus métodos.

Nota: La persistencia no es ni una capacidad ni una propiedad de la POO, no tiene nada que ver con el paradigma en sí, sólo es el mecanismo que se usa para persistir información de un determinado tipo (como puede ser serializar, guardar los datos en una tabla, en un archivo plano, etc).

Desde el punto de vista de la persistencia, se podrían clasificar los objetos en:

- **Transitorios**: Cuyo tiempo de vida depende directamente del ámbito del proceso que los instanció.
- **Persistentes**: Cuyo estado es almacenado en un medio secundario para su posterior reconstrucción y utilización, por lo que su tiempo de vida es independiente del proceso que los instanció.

La persistencia permite al programador **almacenar, transferir y recuperar el estado de los objetos**. Para esto existen varias técnicas:

- Serialización en ficheros.
- Serialización en ficheros XML
- Bases de datos orientadas a objetos.
- Bases de datos SQL.
- Motores de persistencia.

Serialización

La *serialización* (*marshalling*) consiste en un proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato más legible como XML o JSON, entre otros. La serie de bytes o el formato legible pueden ser usados para crear un nuevo objeto idéntico en todo al original, con el mismo estado interno (por tanto, el nuevo objeto es un clon del original).

La serialización es un mecanismo ampliamente usado para:

- Transportar objetos a través de una red.
- Hacer persistente un objeto en un archivo o base de datos

- Distribuir objetos idénticos a varias aplicaciones o localizaciones.

La serialización proporciona una serie de posibilidades:

- Es un método de persistencia de objetos que permite escribir sus propiedades a un archivo de texto.
- Es una forma de realizar llamadas a procedimiento remoto, por ejemplo, como en SOAP.
- Es una manera de distribuir objetos, especialmente en los componentes software, tales como COM, CORBA, etc.
- Es una manera de detectar cambios en variables a lo largo del tiempo.

Estado de un objeto

El estado de un objeto viene dado, básicamente, por el valor de sus atributos. Así, serializar un objeto consiste, básicamente, en guardar los valores de sus atributos. Si el objeto a serializar tiene atributos que a su vez son objetos, habrá que serializarlos primero en un proceso recursivo que implica la serialización profunda de todos los niveles de composición o agregación de objetos. También se almacena información relativa a la estructura de anidamiento producida, para poder llevar a cabo la reconstrucción del objeto serializado.

En ocasiones puede interesar que un atributo concreto de un objeto no sea serializado. Esto se puede conseguir utilizando el modificador **`transient`**, que informa a la JVM de que no nos interesa mantener el valor de ese atributo para serializarlo o hacerlo persistente.

Ejemplo: Atributo no serializado

```
public class MiFecha implements Serializable {
	protected int n;
	protected Date fecha;
	protected transient long s;
	. . .
}
```

En este ejemplo, los atributos `n` y `fecha` serán incluídos en la secuencia de bytes resultante de serializar un objeto de clase **`MiFecha`**. El atributo `s` no será incluido, por tener el modificador **`transient`**.

Interfaz Serializable

Un objeto serializable es un objeto que se puede convertir en una *secuencia de bytes*.

Para que un objeto sea serializable basta con que su clase, o una clase base de ésta, implemente la interfaz `java.io.`**`Serializable`** o su derivada `java.io.`**`Externalizable`**. Esta interfaz no define ningún método, simplemente se usa para *marcar* aquellas clases cuyas instancias pueden ser convertidas en *secuencias de bytes* y posteriormente reconstruidas.

Muchas clases en las API's de Java implementan **`Serializable`**:

- ❏ clases de utilidad, como `java.util.`**`Date`**.
- ❏ Todas las clases de componentes de *Swing GUI*
- ❏ Clases comunes como **`String`**, **`Vector`** o **`ArrayList`**, etc.

Para serializar un objeto no hay más que declarar su clase como **`Serializable`**, sin más, y el sistema de ejecución de Java se encarga de hacer la serialización de forma automática.

Si se intenta serializar un objeto de una clase que no implementa la interfaz **`Serializable`**, se producirá una **`NotSerializableException`** al ejecutar el programa.

Interfaz Externalizable

La interfaz en `java.io.`**`Externalizable`** permite obtener un mayor control sobre el proceso de serialización y reconstrucción de los objetos. Esta interfaz define dos métodos, `writeExternal()` y `readExternal()`, que se encargan de serializar y reconstruir un objeto, respectivamente.

La serialización mediante **`Externalizable`** requiere de un mayor cuidado. De forma automática, sólo se guarda información relativa a la identidad de la clase del objeto que se está serializando. No se guarda automáticamente ni su estado ni información relativa a las clases base de las que herede. Por ello, en la implementación de `writeExternal()` hay que guardar explícitamente el estado de los atributos, incluidos los heredados. A la hora de implementar `writeExternal()` y `readExternal()`, se ha de tener muy en cuenta la serialización de las clases superiores en la jerarquía de herencia y coordinar la implementación de estos métodos con la de los mismos métodos en clases superiores.

Constructor por defecto y deserialización

Para que el proceso de recreación de un objeto desde un flujo de entrada funcione correctamente es necesario que la clase, que implementa la interfaz **`Serializable`**, disponga del constructor por defecto u omisión; al igual que todas las clases de objetos anidados. Durante la reconstrucción en memoria, se invoca al operador **`new`** y al constructor por defecto para disponer de un objeto donde rellenar los valores obtenidos del flujo de entrada.

serialVersionUID

En el proceso de deserialización se utiliza un número de versión asociado a cada clase **`Serializable`** que sirve para verificar que el emisor y el receptor de un objeto serializado mantienen una compatibilidad en lo que a serialización se refiere con respecto a la clases que tienen cargadas (el emisor y el receptor).

En el proceso de serialización se asocia un `serialVersionUID` a cada clase implicada**.** Si alguna de las clases no especifica su correspondiente `serialVersionUID` explícitamente; se calcula un `serialVersionUID` por defecto teniendo en cuenta diferentes aspectos de la clase afectada.

Persistencia en ficheros con streams de objetos

Las clases **`ObjectInputStream`** y **`ObjectOutputStream`** permiten recibir y enviar objetos serializables a través de un *stream*.

Estos *stream* permiten leer y escribir cualquier objeto de alguna clase que implemente la interfaz **`Serializable`**.

Ejemplo: Persistencia de objetos en fichero

```
import java.io.*;

public class ClaseSerializable implements Serializable {

	public static void main(String args) {

		try {

			FileOutputStream fos = new FileOutputStream("fichero.dat");
			FileInputStream fis = new FileInputStream("fichero.dat");

			ObjectOutputStream out = new ObjectOutputStream(fos);
			ObjectInputStream in = new ObjectInputStream(fis);

			ClaseSerializable o1 = new ClaseSerializable();
			ClaseSerializable o2 = new ClaseSerializable();

			// Escribe el objeto serializado en el fichero
			out.writeObject(o1);
			out.writeObject(o2);
			//. . .

			// Lee bytes desde el fichero y deserializa en el mismo
			orden
			o1 = (ClaseSerializable)in.readObject();
			o2 = (ClaseSerializable)in.readObject();
			//. . .
		}
		catch (FileNotFoundException e) {
		}
		catch (IOException e) {
		}
		catch (ClassNotFoundException e) {
		}
	}
}
```

Al deserializar o recrear el objeto desde un flujo de entrada es necesario especificar la clase de

objeto (**`ClaseSerializable`**) para que el sistema identifique cuál su estructura.

Ejemplo: Persistencia de una lista en fichero

```
import java.io.*;
import java.util.*;

public class PersistenciaLista {

	//Lista de nombres
	public static ArrayList<Nombre> datos = new ArrayList<Nombre>();

	public static void main(String[] args) {

		cargarDatosPrueba();
		guardarDatos();			//en fichero
		recuperarDatos();		//desde fichero
		System.out.println(ListarDatos());
	}

	/**
	*  Carga datos de prueba.
	*/
	public static void cargarDatosPrueba() {
		//datos de prueba
		datos.add(new Nombre("Alberto"));
		datos.add(new Nombre("Benito"));
		datos.add(new Nombre("Carlos"));
		datos.add(new Nombre("Demetrio"));
	}

	/**
	*  Lista los datos.
	*  @return String con los datos
	*/
	public static String ListarDatos() {

		StringBuilder aux = new StringBuilder();
		// Recorre la lista de datos.
		for (Nombre n: datos) {
			aux.append(n.texto + "\n");
		}
		return aux.toString();
	}

	// Persistencia de objetos de una lista serializados en ficheros.

	/**
	 *  Guarda el arraylist de objetos Nombre en el fichero.
	 */
	public static void guardarDatos() {
```

```
		try {
			//1º
			// Configurar el fichero local de salida.
			FileOutputStream fNombres =
					new FileOutputStream("nombres.dat");

			//2º
			// Configurar el flujo de datos (stream) de salida.
			ObjectOutputStream osNombres =
					new ObjectOutputStream(fNombres);

			//3º
			// Volcar el arrayList al fichero.
			osNombres.writeObject(datos);

			osNombres.flush();		//vacía buffer
			osNombres.close();		//cierra flujo (stream)
			fNombres.close();		//cierra fichero

		} //try

		catch (FileNotFoundException e) {
			e.printStackTrace();
			System.out.println("Archivo " + "nombres.dat"
					+ " no encontrado");
		}
		catch (EOFException e) {
			e.printStackTrace();
			System.out.println("Fin de fichero");
		}
		catch (IOException eio) {
			eio.printStackTrace();
			System.out.println("Error de entrada/salida");
		}
		catch (Exception e) {
			e.printStackTrace();
			System.out.println("Error general");
		}
		finally {
		}
	}

	/**
	 *  Recupera el arraylist de objetos Nombre almacenados en fichero.
	 */
	public static void recuperarDatos() {
		try {
			//Previo
			//Limpiar el ArrayList de posible basura
```

```
                    datos.clear();

                    //1º
                    //Configurar el fichero de entrada
                    FileInputStream fNombres =
                                        new FileInputStream("nombres.dat");

                    //2º
                    //Configurar el flujo de datos (stream) de entrada
                    ObjectInputStream isNombres =
                                        new ObjectInputStream(fNombres);

                    //3º
                    //Cargar el arrayList desde el fichero
                    datos = (ArrayList<Nombre>) isNombres.readObject();

                    isNombres.close();              //cierra flujo (stream)
                    fNombres.close();               //cierra fichero

             } //try

             catch (FileNotFoundException enf) {
                    enf.printStackTrace();
                    System.out.println("Archivo " + "nombres.dat"
                                        + " no encontrado");
             }
             catch (EOFException eef) {
                    eef.printStackTrace();
                    System.out.println("Fin de fichero");
             }
             catch (IOException eio) {
                    eio.printStackTrace();
                    System.out.println("Error de entrada/salida");
             }
             catch (Exception e) {
                    e.printStackTrace();
                    System.out.println("Error general");
             }
             finally {
             }
      }

} //class


/**
*  Clase auxiliar para el ejemplo.
*/
class Nombre implements Serializable {
```

```
	//Atributo
	public String texto;

	//constructor
	public Nombre(String n) {
		texto = n;
	}

} //class
```

Bases de datos orientadas a objetos

Las bases de datos orientadas a objetos (BDOO) están, como su nombre indica especialmente diseñadas para trabajar con datos de tipo objeto mientras que las bases de datos tradicionales la filosofía es totalmente distinta. Estas bases de datos tradicionales (generalmente relacionales) siguen los modelos clásicos de datos mientras que en los modelos Orientados a Objetos los datos manejados por la base de datos serán clases y objetos.

Uno de los problemas que se encuentran los programadores en los lenguajes orientados a objetos es la necesidad de almacenar y recuperar los datos de una forma eficiente y sencilla. Si se utilizan modelos relacionales se pierde parte de las ventajas de trabajar con Orientación a Objetos puesto que hay que hacer una transformación entre objetos y datos relacionales (y viceversa). Esta diferencia de paradigmas fue la que impulsó a crear los sistemas gestores de bases de datos orientados a objetos (SGBDOO) para evitar las dificultades del paso del modelo de objetos al modelo relacional, se buscaba la eficiencia y la sencillez. ¿Por qué perder tiempo en rehacer las estructuras de datos (objetos) cuando se pueden almacenar y recuperar directamente?

Bases de datos orientadas a objetos comerciales

A continuación se enumeran algunas Bases de Datos Orientadas a Objetos disponibles en el mercado:

- Objectivity/DB. Es una Base de Datos Orientada a Objetos que ofrece soporte para Java, C++, Python y otros lenguajes. Ofrece un alto rendimiento y escalabilidad. No es un producto libre aunque ofrecen versiones de prueba durante un periodo determinado.
- db4o. Es una Base de Datos Open Source para Java y .NET. Se distribuye bajo licencia GPL. Fue adquirida en 2008 por la empresa Versant Corporation. Después dejó de evolucionar el producto. La última versión disponible para Java es la db4o-8.0.276.16149 Se puede descargar.

- Intersystems Caché. Es una Base de Datos Orientada a Objetos que ofrece soporte para Java, C++, .NET, etc. Se vanaglorian de poder manejar grandes volúmenes de información y ejecutar sentencias SQL de forma más rápida que algunas bases de datos relacionales. Todo esto con un consumo mínimo de recursos y de hardware.
- EyeDB. Sistema gestor de Bases de Datos Orientado a Objetos (OODBMS) basado en la especificación ODMG el cual está desarrollado y soportado por la compañía francesa SYSRA. Proporciona un modelo avanzado de objetos (herencia, colecciones, array, métodos, triggers, constraints, etc.), un lenguaje de definición de objetos basado en ODMG

ODL, un lenguaje de manipulación y consulta de datos basado en ODMG OQL e interfaces de programación para C++ y Java. Se distribuye bajo la licencia GNU (GNU lesser General Public License). Es un software libre.

Características de las bases de datos orientadas a objetos

Las características de las Bases de Datos Orientadas a Objetos son análogas a los lenguajes de programación OO. Algunas de las características relevantes de una Base de Datos Orientada a Objetos son las siguientes:

- Se entienden las Bases de Datos OO como un sistema de modelado del mundo real. Cada entidad del mundo real será un objeto en la base de datos.
- Los objetos tienen un identificador único que los identifica y los diferencia de los demás objetos del mismo tipo.
- Eso implica por ejemplo, que cada vez que se quiera modificar un objeto se tenga que recuperar de la base de datos, modificar y almacenar nuevamente. Esta operación es totalmente diferente en los SGBDR.
- Pueden existir objetos con la misma información pero con diferente identidad. Es más, cuando se modifican los valores de los atributos, el objeto sigue siendo el mismo.
- Es posible almacenar objetos complejos en la base de datos sin que por ello haya que realizar operaciones especiales sobre la base de datos.
- El concepto de herencia se mantiene incluso en la base de datos.
- Es el usuario el que modela los objetos de la base de datos y etiqueta los atributos y métodos que son visibles en la interfaz del objeto y cuáles no.
- El SGBDOO se encarga de acceder a los miembros de los objetos sin necesidad de escribir métodos para acceder a ellos.
- Acceso rápido a los datos dado que no hace falta realizar joins de tablas.
- Control de versiones. Algunos SGBDOO permiten un control de versiones.
- Implantación de conceptos del modelo OO como (polimorfismo, sobrecarga, sobreescritura, etc.).

Por lo tanto, dadas las características anteriores las Bases de Datos Orientadas a Objetos se hacen más apropiadas cuando tenemos una gran cantidad de tipos de datos diferentes, objetos con comportamientos avanzados o con un gran número de relaciones entre ellos.

- Entre las ventajas que obtenemos al trabajar con las Bases de Datos Orientadas a Objetos se encuentran las siguientes:
- La primera y fundamental es el no tener que reensamblar los objetos cada vez que se accede a la base de datos.
- El resultado de las consultas son objetos por lo tanto la velocidad de procesamiento se aumenta.
- Cuando cambia un objeto la forma de actualizarlo en la base de datos es simplemente volviendo a guardar. Esta acción suele ser una secuencia simple de comandos en el código.
- La reutilización que es una de las características de los lenguajes de programación orientados a objetos se mantiene con lo que se mejoran los costes de desarrollo.
- El acceso a la información a través de objetos en una aplicación desarrollada como tal es más natural que hacerlo a través de tablas y filas.
- El control de acceso y concurrencia se facilita enormemente dado que se puede bloquear el acceso a ciertos objetos incluso en una jerarquía completa de objetos.

- Estos sistemas funcionan de forma eficiente en entornos cliente/servidor y arquitecturas distribuidas.
- No hace falta redefinir las relaciones entre objetos pues ya existen en el modelo de objetos y la base de datos se encarga de hacer persistente lo ya diseñado.

Por otra parte las limitaciones de las Bases de Datos Orientadas a Objetos serán las siguientes:

- En las bases de datos relacionales se dispone del lenguaje SQL que es un estándar bastante asentado y fuerte. En las bases de datos orientadas a objetos esto no ocurre.
- La estructura de las bases de datos relacionales es simple y fácil de entender al contrario que la de las bases de datos orientadas a objetos.
- Existen aplicaciones que aunque están escritas con lenguajes orientados a objetos, en ciertas ocasiones es más eficiente almacenar los datos en bases de datos relacionales debido a las consultas que se van a realizar sobre ellas.
- Se reduce la velocidad de acceso debido a que la Base de Datos Orientada a Objetos tiene que tener en cuenta las relaciones de herencia entre clases.
- En ocasiones aunque se gana en simplicidad en el manejo de la base de datos desde los lenguajes de programación orientados a objetos, cuando se trata de realizar consultas complejas resulta más adecuado una base de datos relacional accedida mediante SQL.

Bases de datos db4o

El sistema de base de datos db4o , es una base de datos con licencia GPL. Db4o es una verdadera base de datos de objetos; el estado de los objetos, así como la estructura y las relaciones son almacenados en la base de datos sin importar los niveles de anidamiento de los objetos.

db4o es una base de datos de código nativo Java abierto. Persistir cualquier objeto con una línea de código. Permite la replicación orientada a objetos, consultas nativas, el reconocimiento automático de esquema y 350K de huella. Es ideal para uso incrustado, por ejemplo, en software que se ejecuta en los dispositivos móviles o médicos, en software empaquetado, y para sistemas de tiempo real.

Existen distintas distribuciones de la base de datos db4o para Java, .NET y Mono. Cualquier base de datos creada con cualquiera de estos lenguajes es intercambiable entre sí. Eso quiere decir que se puede utilizar una base de datos creada con .NET desde Java y viceversa.

Licencia de db4o

La licencia de db4o, al igual que algunas otras herramientas GPL tiene una licencia dual; se puede utilizar sin problema para el desarrollo, uso interno o asociada a otras herramientas GPL. No obstante, si se quiere utilizar en software comercial se necesitará adquirir una licencia de db4o. El propietario actual de db40 es Versant Corporation que publica la siguiente información sobre licencia y soporte: http://supportservices.actian.com/versant/default.html

Db4o es una base de datos orientada a objetos nativa de Java. La distribución es un único zip y su contenido es el siguiente:

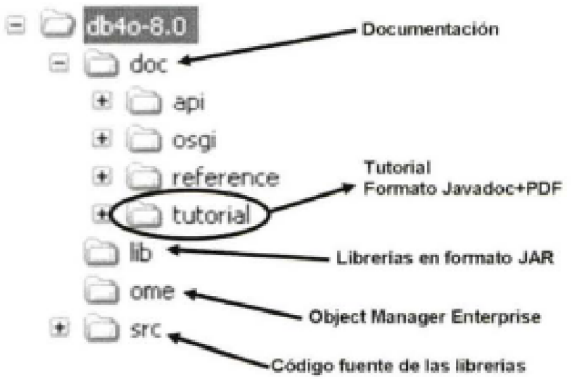

Figura 9.1. Contenido de la distribución db4o versión 8.0

El motor de la base de datos Db4o tiene varias configuraciones para su versión 8 .0 :

- **db4o-8.0-core*.jar**. Archivos -**core**, contienen el núcleo del motor.
- **db4o-8.0-cs*.jar**. Archivos -**cs**, contienen la versión cliente/servidor.
- **db4o-8.0-optional *.jar**. Archivos **-optional**, añaden funcionalidad avanzada al motor de base de datos.
- **db4o-8.0-all *.jar**. Contienen todas las características anteriores. Instalación completa.

La base de datos también está disponible para las diferentes versiones de JDK. Por ejemplo para la instalación completa existen las siguientes versiones:

- **db4o-8.0-all-java1.1.jar**. Funciona con la mayoría de JDKs dado que se ha escrito para proporcionar máxima retrocompatibilidad. Sería compatible hasta los JDK 1. 1.x.
- **db4o-8.0-java1.2.jar**. Desarrollado para los JDK entre las versiones 1.2 y 1.4.
- **db4o-8.0-java5.jar**. Desarrollado para los JDK 5 Y 6.

La instalación consiste en copiar el motor de base de datos que son las clases necesarias para hacer que funcione la API en toda su extensión y alguna aplicación para visualizar los datos con los que se está trabajando.

Instalación de motor de la base de datos

La instalación consistirá en colocar uno de los ficheros **db40-core* .jar** dentro del directorio donde apunte la variable `CLASSPATH`.

Instalación dentro de un lDE

En ocasiones, puede interesar instalar la base datos en un IDE y ubicarla dentro de un directorio `./lib` en el directorio de trabajo . En ese caso habrá que configurar el IDE para indicar que añada la librería (el JAR) db40 situada en el directorio `./lib` al proyecto actual.

El visor de objetos Object Manager Enterprise

El *Object Manager* es una herramienta disponible en el zip de la instalación; se se incluye un plugin para Eclipse que permite ejecutar el *Object Manager*.

La API (Application Program Interface)

La documentación de la API viene en formato *JavaDoc* en el directorio `/doc/api` del zip descargado. Una de las interfaces más importantes es la siguiente:

`com.db4o.`**`ObjectContainer`**

Este contenedor puede representar una base de datos en modo embebido en la aplicación o un cliente de un servidor db4o. Esta **`interface`** proporciona métodos para almacenar, consultar y borrar objetos así como realizar transacciones sobre la base de datos.

Cada **`ObjectContainer`** representa una transacción. Todas las operaciones realizadas en la base de datos se hacen en modo transaccional de tal manera que cuando se haga `commit()` o `rollback()` automáticamente se inicia la siguiente transacción.

Cada **`ObjectContainer`** mantiene sus propias referencias a los objetos instanciados y almacenados.

Operaciones básicas con la base de datos

Una clase de ejemplo para pruebas puede ser la siguiente:

```
public class Alumno {
	private String nombre;
	private int edad;
	private double nota;
	public Alumno() {
		this.nombre = null;
		edad = 0;
		nota = 0;
	}

	public Alumno(String n, int e) {
		this.nombre = n;
		this.edad = e;
		this.nota = -1; //nota no establecida
	}

	public alumno(String nom, int e, double not) {
		this.nombre = nom;
		this.edad = e;
		this.nota = not;
	}

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

	public String getNombre() {
		return this.nombre;
	}

	public void setNota(double n) {
```

```
		this.nota = n;
	}

	public double getNota() {
		return this.nota;
	}

	public void setEdad(int e) {
		this.edad = e;
	}

	public int getEdad() {
		return this.edad;
	}

	public String toString() {
		if (this.nota != -1)
			return this.nombre + "(" + this.edad + ") Nota:" +this.nota;
		return this.nombre + "(" + this.edad + ")";
	}
}
```

Como puede comprobarse, la clase anterior representa el concepto de Alumno y tiene los métodos mínimos para poder trabajar con ella. A continuación, se pueden ver las operaciones básicas que se pueden realizar sobre esta base de datos.

Crear/acceder a la base de datos

Hay una secuencia básica de trabajo con la base de datos:

Apertura de la BD (conexión) ---> Realizar operaciones ---> Cerrar la BD (desconexión)

Un ejemplo sería el siguiente:

```
ObjectContainer bd =
Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),"alumnos.db4o") ;
try {
     //Realizar operaciones o llamadas a métodos

}
finally {
    bd.close();
}
```

Métodos de apertura y cierre de una base de datos db4o

`Db40Embedded.openFile(Db4oEmbedded .newConfiguration(), "alumnos.db4o");`	El método `openfile()` abre una base de datos y si no existe la crea . Recibe dos parámetros, uno de tipo `Configuration` que se puede obtener realizando la llamada `Db4oEmbedded. newConfiguration()`.

La interfaz `Configuration` contiene métodos para poder configurar db4o.

El segundo parámetro es el nombre del fichero donde se va a alojar la base de datos.

```
bd.close();
```

Se le insta al objeto de tipo `ObjectContainer` a cerrar la base de datos.

Si al abrir la base de datos ésta no existe se creará.

Una base de datos solamente se puede abrir una vez. Si se hacen varias aperturas de la base de datos generará una excepción del tipo `DatabaseFileLockedException`.

Almacenar objetos

Para almacenar objetos en la base de datos basta con llamar al método `store()` del objeto bd creado de tipo **`ObjectContainer`**. Un ejemplo de método que almacena tres objetos **`Alumno`** es el siguiente:

```
public static void almacenarAlumnos(ObjectContainer bd) {
	Alumno a1 = new Alumno("Juan Gámez", 23, 8.75);
	bd.store(a1);
	System.out.println(a1.getNombre() + "- Almacenado");
	Alumno a2 = new Alumno("Emilio Anaya", 24, 6.25);
	bd.store(a2);
	System.out.println(a2.getNombre() + "- Almacenado");
	Alumno a3 = new Alumno("Ángeles Blanco", 26, 7) ;
	bd.store(a3);
	System.out.println(a3.getNombre() + "- Almacenado");
}
```

```
}
```

Salida:

```
Juan Gámez - Almacenado

Emilio Anaya - Almacenado

Ángeles Blanco - Almacenado
```

Recuperar objetos de la base de datos

Cuando se usan las bases de datos relacionales las recuperaciones de datos de la base de datos se hacen utilizando un lenguaje llamado SQL (Structured Query Language). Este lenguaje es específico para bases de datos relacionales. Con las bases de datos orientadas a objetos se utilizan

otros lenguajes.

Para mostrar el resultado de las consultas realizadas a la base de datos vamos a necesitar un nuevo método, `mostrarResultado()`que toma como parámetro un **`ObjectSet`** (conjunto de objetos) y lo recorre mostrando uno a uno los datos recuperados. El código del método es el siguiente:

```
public static void mostrarResultado(ObjectSet res) {
    System.out.println("Recuperados " + res.size() + " Objetos");
    while (res.hasNext()) {
        System.out.println(res.next()) ;
    }
}
```

Para recuperar todos los alumnos de la base de datos hay pasa un objeto vacío de la clase **`Alumno`** (datos de tipo **`String`** a **`null`** y campos numéricos a `0`) que le indica a la base de datos que deberá recuperar todos los objetos.

El siguiente código recupera todos los objetos **`Alumno`** de la base de datos:

```
public static void muestraAlumnos(ObjectContainer bd) {
     Alumno a = new Alumno(null, 0, 0);
     ObjectSet res = bd.queryByExample(a);
     mostrarResultado(res);
}
```

Salida:

```
Recuperados 3 Objetos

Juan Gámez(23) Nota: 8.75

Emilio Anaya(24) Nota: 6.25

Ángeles Blanco(26) Nota: 7
```

Otras variantes de este tipo de consultas sería recuperar un alumno concreto por su nombre o un alumno cuya edad sea, por ejemplo, **26**. En este caso se le pasa un objeto **`Alumno`** cuyo campo edad tenga el valor 26:

```
public static void muestraAlumnos(ObjectContainer bd) {
     Alumno a = new Alumno(null, 26, 0);
     ObjectSet res = bd .queryByExample(a);
     mostrarResultado(res);
}
```

Salida:

```
Recuperados 1 Objetos

Ángeles Blanco(26) Nota: 7
```

Actualizar objetos en la base de datos

Actualizar un objeto es muy fácil. Basta con modificarlo y llamar al método `store()` para que se actualice en la base de datos. Veamos un ejemplo:

```
public static void actualizarNotaAlumno(ObjectContainer bd, String nombre,
                                        double nota) {
      ObjectSet res = bd.queryByExample(new alumno(nombre, 0, 0));
      Alumno a = (Alumno)res.next();
      a.setNota(nota);
      bd.store(a);
      muestraAlumnos(bd);
}
```

Al realizar la llamada al método:

```
actualizarNotaAlumno(bd, "Emilio Anaya", 9.5);
```

Salida:

```
Recuperados 3 Objetos

Juan Gámez(23) Nota: 8.75

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

Se modifica la nota de Emilio Anaya por el valor 9,5.

Hay que tener en cuenta que los objetos para poder ser actualizados deben de haber sido insertados o recuperados en la misma sesión, si no se añadirá otro objeto en vez de actualizarse. Db4o, para modificar un objeto debe conocerlo previamente.

Borrar objetos de la base de datos

El siguiente método borrará un alumno de la base de datos.

```
public static void borrarAlumnoNombre(ObjectContainer bd, String nombre) {
    ObjectSet res = bd.queryByExample(new Alumno(nombre , 0, 0));
    alumno a = (alumno)res.nex();
    bd.delete(a);
    muestraAlumnos(bd);
}
```

Al realizar la llamada al método:

```
borrarAlumnoNombre(bd, "Juan Gámez");
```

Salida:

```
Recuperados 2 Objetos

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

Es importante tener en cuenta, como ya se ha dicho, que los objetos para poder ser borrados, al igual que para ser actualizados deben de haber sido insertados o recuperados en la misma sesión. Aunque se proporcione un objeto con los mismos datos no es suficiente.

Consultas a la base de datos

Db4o tiene la posibilidad de consultar la base de datos mediante tres tipos de sistemas:

- **Query By Example (QBE)**. Es el sistema ya visto anteriormente.
- **Native Queries (NQ)**. Son consultas nativas. Es la interfaz principal de la base de datos y aconsejado por los desarrolladores de db4o.
- **SODA (Simple Object Data Access)**. Es la API interna. Se puede utilizar para una mayor retrocompatibilidad o para generar consultas dinámicamente. Es mucho más potente que las dos anteriores y mucho más rápida dado que los dos tipos de consultas anteriores (QBE y NQ) tienen que ser traducidas a SODA para ejecutarse.

QBE es la forma más básica de consultar la base de datos. Es sencilla porque funciona presentando un ejemplo y se recuperarán los datos que coincidan con el mismo; pero esto hace que tenga muchas limitaciones. Algunas de estas limitaciones al utilizar QBE son las siguientes:

- Al contrario que con otros lenguajes de consulta no se pueden realizar expresiones avanzadas (por ejemplo operadores AND, OR, NOT, etc.).
- Hay que proporcionar un ejemplo con las limitaciones que ello conlleva.
- No se puede preguntar por objetos cuyo valor de un campo numérico sea cero, texto vacío o algún campo que sea **`null`**.
- Se necesita un constructor para crear objetos con campos no inicializados.

Librería API SODA

En este apartado vamos a ver cómo funcionan las consultas utilizando la librería SODA (Simple Object Database Access). Esta librería presenta la ventaja que las consultas se ejecutan de la manera más rápida y permite generar una consulta dinámica (las consultas dinámicas son las consultas que solo se conocen en tiempo de ejecución).

Para crear una consulta expresada en SODA se necesita un objeto de tipo Query el cual puede ser generado llamando al método `query()` del objeto de tipo **`ObjectContainer`**. Una vez creado este objeto **`Query`** se le van añadiendo *Constraints* (restricciones) para ir modelando el resultado que se quiere obtener. Una vez que ya se tiene modelada la consulta se ejecuta la consulta llamando al método `execute()` del objeto tipo **`Query`**.

Un método que genera un listado de todos los objetos alumno de la base de datos:

```
public static void consultaSODAAlumnos(ObjectConLainer bd) {
    Query query = bd.query();
    query.constrain(Alumno.class);
    ObjectSet result = query.execute();
    mostrarResultado(result);
}
```

Al ejecutar el método `consultaSODAAlumnos(bd);`

Salida:

```
Recuperados 2 Objetos

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

La consulta anterior de una forma gráfica es un grafo como el siguiente:

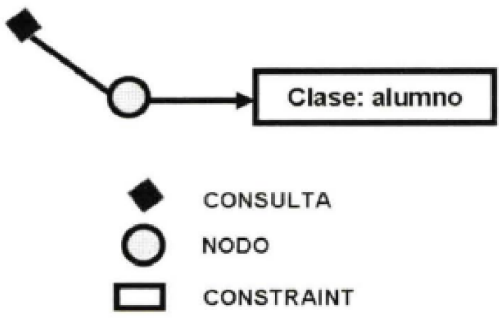

Figura 9.3. Consulta SODA I

Lo que se va a construir es una consulta con una serie de nodos que filtran los objetos candidatos.

Constrain: Limita el número de objetos devueltos en la ejecución de una query.

Query: referencias a un nodo en SODA.

Supongamos que queremos tener todos los alumnos que se llamen `"Emilio Anaya"`, en ese caso deberemos *descender* o entrar en el objeto y llegar al atributo `nombre` y añadirle una `constraint` que establezca nombre a `"Emilio Anaya"`. El grafo de la consulta que se quiere realizar sería el siguiente:

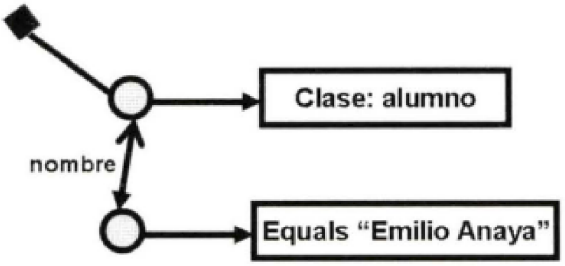

Figura 9.4. Consulta SODA II

```
public static void consultaSODAEmilio(ObjectContainer bd) {
    Query query = bd.query();
    query.constrain(Alumno.class);
    query.descend("nombre").constrain("Emilio Anaya");
    ObjectSet result = query.execute();
    mostrarResultado(resul);
}
```

Al ejecutar el método `consultaSODAEmilio(bd);`

Salida:

```
Recuperados 1 Objetos

Emilio Anaya(24) Nota: 9.5
```

Algo parecido pasaría si en vez del nombre queremos seleccionar aquellos alumnos que tienen 23 años. El grafo sería muy parecido:

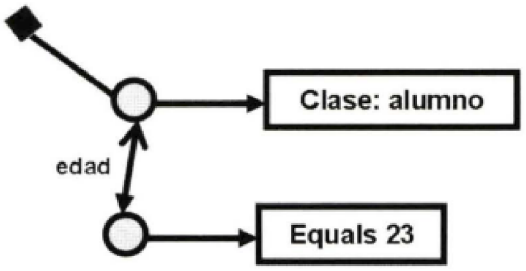

Figura 9.5. Consulta SODA III

```
public static void consultaSODAAlumnoEdad(ObjeetContainer bd,
int edad) {
    Query query = bd.query();
    query.constrain(Alumno.class);
    query.descend("edad").constrain(edad);
```

```
    ObjectSet result = query.execute();
    mostrarResultado(result);
}
```

Al realizar la llamada al método:

```
consultaSODAAlumnoEdad(bd, 24);
```

Salida:

```
Recuperados 1 Objetos

Emilio Anaya(24) Nota: 9.5
```

Supongamos que queremos seleccionar aquellos alumnos que no tiene 23 años. El grafo sería:

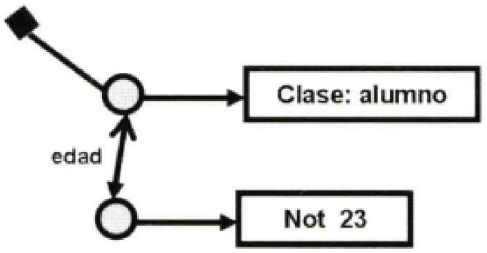

Figura 9.6. Consulta SODA IV

```
public static void consultaSODAAlumnoNoEdad(ObjeetContainer bd,
int edad) {
    Query query = bd.query();
    query.constrain(Alumno.class);
    query.descend("edad").constrain(edad).not();
    ObjectSet result = query.execute();
    mostrarResultado(result);
}
```

Al realizar la llamada al método:

```
consultaSODAAlumnoNoEdad(bd, 23);
```

Salida:

```
Recuperados 1 Objetos

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

Supongamos ahora que queremos conocer los alumnos mayores de 23 años y menores de 25. El grafo y el código serían los siguientes:

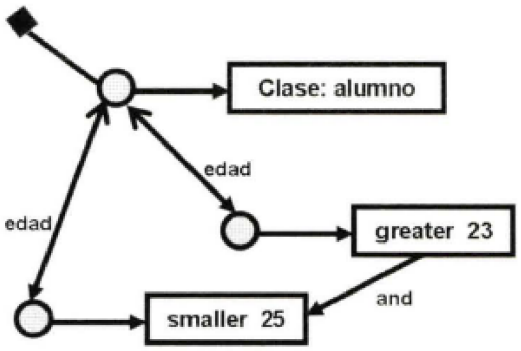

Figura 9.7. Consulta SODA V

```
public static void consultaSODAAlumnoRangoEdad(ObjeetContainer
bd, int eMinima, int eMaxima) {
    Query query = bd.query();
    query.constrain(Alumno.class);
    query.descend("edad").constrain(eMinima).greater();
    query.descend("edad").constrain(eMaxima).smaller();
    ObjectSet result = query.execute();
    mostrarResultado(result);
}
```

Al realizar la llamada al método:

```
consultaSODAAlumnoRangoEdad(bd, 23, 27);
```

Salida:

```
Recuperados 2 Objetos

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

Otra consulta podría ser los alumnos cuya nota sea menor de 7 o edad mayor de 25. El grafo y el código completo de la clase para probarlo todo serían los siguientes:

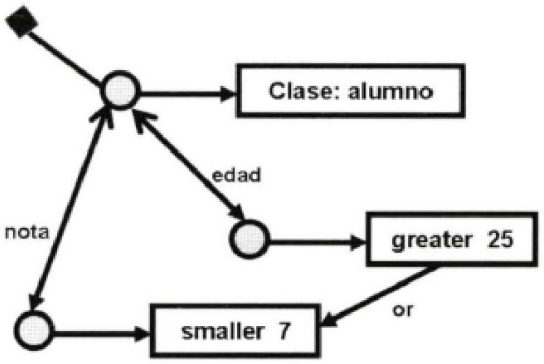

Figura 9.8. Consulta SODA VI

```
import java.io.*;
import com.db4o.*;
import com.db4o.query.*;

public class testSODA  {

    public static void main(String[] args) {
         new File("alumnos.db4o").delete();
         ObjectContainer bd =
b4oEmbedded.openFile(Db4oEmbedded
            .newConfiguration(), "alumnos.db4o");
         try {
              almacenarAlumnos(bd);
              consultaSODA(bd);
         }
         catch(Exception e) {
              e.printStackTrace();
         }
         finally {
             bd.close();
         }
    }

    public static void mostrarResultado(ObjectSet  res) {
         System.out.println("Recuperados " + res.size()+"
Objetos");
         while(res.hasNext()) {
            System.out.println(res.next());
         }
    }

    public static void almacenarAlumnos(ObjectContainer bd) {
        alumno a1 = new alumno("Juan Gámez",23,8.75);
        bd.store(a1);
        System.out.println(a1.getNombre()+" Almacenado");
```

```
        alumno a2 = new alumno("Emilio Anaya",24,6.25);
        bd.store(a2);
        System.out.println(a2.getNombre()+" Almacenado");
        alumno a3 = new alumno("Angeles Blanco",26,7);
        bd.store(a3);
        System.out.println(a3.getNombre()+" Almacenado");
    }

    public static void consultaSODA(ObjectContainer bd) {
        Query query=bd.query();
        query.constrain(alumno.class);
        Constraint constr = query.descend("nota")
                            .constrain(7).smaller();

query.descend("edad").constrain(25).greater().or(constr);
        ObjectSet result=query.execute();
        mostrarResultado(result);
    }

}
```

Al realizar la llamada al método:

```
consultaSODA(bd);
```

Salida:

```
Recuperados 2 Objetos

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7
```

Serialización de objetos Java en XML

JAXB (Java Architecture for XML Binding) proporciona una API que permite mapear (mapping) automáticamente entre documentos XML y objetos Java. Con este framework se puede serializar un objeto Java en un fichero XML persistiendo el estado del objeto. Lógicamente, también se puede crear un objeto Java a partir de un documento XML recuperando el estado que tenía dicho objeto al ser serializado.

Ejemplo: Persistencia en XML

```
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlType
class Localidad {
    private String nombre;
    private int cp;
```

```
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getCp() {
        return cp;
    }
    public void setCp(int cp) {
        this.cp = cp;
    }
}


@XmlRootElement
class Provincia {
    private String nombre;
    private Localidad[] localidad;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public Localidad[] getLocalidad() {
        return localidad;
    }
    public void setLocalidad(Localidad[] localidad) {
        this.localidad = localidad;
    }
}
```

Se utilizan dos anotaciones relacionadas con XML.

- `@XmlType` Indica a JAXB que genere un tipo de dato XML schema a partir de un tipo de dato Java.
- `@XmlRootElement` Indica a JAXB que genere un fichero XML (la raíz) a partir de una clase Java.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

public class JAXBSerializationExample {
```

```
	private static final String PROVINCIA_DAT_FILE = "provincia.dat";

	public static void main(String[] args) throws JAXBException, IOException{

		JAXBContext context = JAXBContext.newInstance(Provincia.class);
		Marshaller marshaller = context.createMarshaller();
		marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

		Provincia provincia = fillProvincia();

		//Mostramos el documento XML generado en la salida estandar
		marshaller.marshal(provincia, System.out);

		FileOutputStream fos = new FileOutputStream(PROVINCIA_DAT_FILE);
		//guardamos el objeto serializado en un documento XML
		marshaller.marshal(provincia, fos);
		fos.close();

		Unmarshaller unmarshaller = context.createUnmarshaller();
		//Deserealizamos a partir de un documento XML
		Provincia provincaAux = (Provincia) unmarshaller.unmarshal(new
File(PROVINCIA_DAT_FILE));
		System.out.println("********* Provincia cargado desde fichero
XML***************");
		//Mostramos por linea de comandos el objeto Java obtenido
		//producto de la deserialziacion
		marshaller.marshal(provincaAux, System.out);

	}

	private static Provincia fillProvincia(){

		String[] nombreLocalidad = {"Madrid", "Coslada"};
		int[] cpLocalidad = {28028, 28820};
		Localidad[] localidades = new Localidad[2];

		for (int i=0; i<2; i++){
			localidades[i] = new Localidad();
			localidades[i].setCp(cpLocalidad[i]);
			localidades[i].setNombre(nombreLocalidad[i]);
		}

		Provincia provincia = new Provincia();
		provincia.setNombre("Madrid");
		provincia.setLocalidad(localidades);

		return provincia;
	}
}
```

Ejercicios

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- Wikipedia. *Persistencia de objetos y serialización.* [en línea] http://es.wikipedia.org/wiki/Persistencia_de_objetos http://es.wikipedia.org/wiki/Serializaci%C3%B3n
- Wikipedia. *Object database.* [en línea]
- https://en.wikipedia.org/wiki/Object_database
- Documentos y ejemplos DB4o. [en línea] http://www.epidataconsulting.com/tikiwiki/tiki-index.php?page=db4o
- Blog. [en línea] http://josedeveloper.com/2012/03/04/serializacion-de-objetos-java-en-xml/